UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Bruno Luan de Sousa

Modeling and Predicting Evolution of Object-Oriented Software Quality Internal Attributes

Belo Horizonte
2023

Bruno Luan de Sousa

Modeling and Predicting Evolution of Object-Oriented Software Quality Internal Attributes

Final Version

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Prof.[a] Mariza Andrade da Silva Bigonha
Co-Advisor: Prof.[a] Kecia Aline Marques Ferreira, Prof.[a] Glaura da Conceição Franco

Belo Horizonte
2023

Sousa, Bruno Luan de.

S725m Modeling and predicting evolution of object-oriented software quality internal attributes [manuscrito] / Bruno Luan de Sousa – 2023.
1 recurso online (132 f. il, color.)

Orientadora: Mariza Andrade da Silva Bigonha.
Coorientadora: Kecia Aline Marques Ferreira
Coorientadora: Glaura da Conceição Franco.

Tese (doutorado) – Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f.108 -123.

1. Computação - Teses. 2, Engenharia de Software - Teses. 3. Software - Desenvolvimento - Teses. 4. Séries temporais - Teses.I. Bigonha, Mariza Andrade da Silva. II. Ferreira, Kecia Aline Marques. III. Franco, Glaura da Conceição. IV. Universidade Federal de Minas Gerais; Instituto de Ciências Exatas, Departamento de Ciência da Computação. V.Título.

CDU 519.6*32.(043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa CRB 6ª Região nº 1510

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

MODELING AND PREDICTING EVOLUTION OF OBJECT-ORIENTED SOFTWARE QUALITY INTERNAL ATTRIBUTES

BRUNO LUAN DE SOUSA

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores(a):

Profa. Mariza Andrade da Silva Bigonha - Orientadora
Departamento de Ciência da Computação - UFMG

Profa. Glaura da Conceição Franco - Coorientadora
Departamento de Estatística - UFMG

Profa. Kecia Aline Marques Ferreira - Coorientadora
Departamento de Computação - CEFET-MG

Prof. Guilherme Horta Travassos
Programa de Engenharia de Sistemas e Computação - COPPE - UFRJ

Prof. José Carlos Maldonado
Instituto de Ciências Matemáticas e de Computação - USP

Prof. André Cavalcante Hora
Departamento de Ciência da Computação - UFMG

Prof. Eduardo Magno Lages Figueiredo
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 29 de agosto de 2023.

Documento assinado eletronicamente por **Mariza Andrade da Silva Bigonha**, **Professora do Magistério Superior**, em 19/10/2023, às 09:45, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Andre Cavalcante Hora**, **Professor do Magistério Superior**, em 20/10/2023, às 10:09, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Eduardo Magno Lages Figueiredo**, **Professor do Magistério Superior**, em 23/10/2023, às 09:12, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Glaura da Conceição Franco**, **Coordenador(a)**, em 24/10/2023, às 10:58, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Kecia Aline Marques Ferreira**, **Usuário Externo**, em 25/10/2023, às 09:31, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Guilherme Horta Travassos**, **Usuário Externo**, em 09/11/2023, às 16:32, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **José Carlos Maldonado**, **Usuário Externo**, em 13/11/2023, às 12:15, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufmg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **2696757** e o código CRC **E0F49FA0**.

Acknowledgments

This work would not have been possible without the support of many people.

First, I thank God for always guiding me, illuminating my paths, and never letting me give up even in the most challenging moments I encountered along this journey.

I thank my father, Paulo, and my mother, Irani, for all their love, support, and encouragement dedicated throughout my life. They are examples of dedication and perseverance, providing the foundation for my education.

I thank my fiance, Giovanna, for always being by my side in the difficult moments and for her friendship, affection, comprehension, and patience over this journey.

I thank my advisors, professors Mariza Bigonha, Kecia Ferreira, and Glaura Franco, for the excellent orientation, attention, encouragement, availability, and patience. Since the beginning of the Ph.D. course, they have monitored my evolution and provided the necessary support to overcome my difficulties.

I thank my professors who were willing to donate knowledge, time, and patience to me so that I could evolve.

I thank all my friends and colleagues with whom I have shared knowledge and incredible moments throughout the Ph.D. course.

I thank the Department of Computer Science at UFMG for the opportunity to participate in its Ph.D. program and for providing a course with a high level of excellence.

I thank the CAPES for the financial support provided throughout the initial three years of the Ph.D. course.

I want to express my gratitude to the member of my thesis project defense committee for their valuable contributions – professors Andre Hora (UFMG), Alessandro Garcia (PUC-Rio), Eduardo Figueiredo (UFMG), and José Carlos Maldonado (USP).

I also thank Professor Guilherme Travassos (UFRJ), who kindly agreed to participate in the final defense.

Finally, I thank everyone who has contributed directly or indirectly to the conduction of this work so far and for having encouraged me always to take it forward.

"*Success is the sum of small efforts repeated day after day.*"
(Robert Collier)

Resumo

Evolução do software é um processo natural do ciclo de vida do software que consiste em adaptar, manter e atualizá-lo. Conhecer e compreender como os sistemas de software evoluem pode contribuir para sua gestão e controle de qualidade. Esta tese de doutorado investiga como os sistemas de software evoluem, especificamente em seus atributos internos de qualidade, como coesão e acoplamento. Embora muitos estudos tenham sido realizados sobre a evolução de software, não havia até agora uma visão geral e consolidada do estado da arte sobre modelos para evolução de software. Assim, nosso primeiro passo foi realizar um Mapeamento Sistemático da Literatura (MSL) para identificar o que tem sido produzido a respeito de modelos para evolução de software. Nosso MSL identificou 71 artigos publicados de 1979 a 2022. A análise desses estudos revelou que: (i) os modelos de evolução de software têm sido construídos com base em três categorias: caracterização, descrição e predição; (ii) os principais focos dos modelos têm sido predição de defeitos e mudanças, caracterização da evolução da estrutura interna do software e detalhamento de como propriedades intrínsecas dos sistemas de software evoluem; (iii) pesquisadores têm preferido construir *datasets* próprios para condução de estudos em evolução de software ao invés de utilizar *datasets* existentes; (iv) existem alguns importantes *datasets* disponibilizados publicamente na literatura, porém eles não contêm uma grande quantidade e variedade de sistemas e estão desatualizados; e (v) as principais técnicas usadas para criar modelos são Aprendizado de máquina, técnicas de Regressão, ARIMA e técnicas Baseada em Grafos. Esta tese de doutorado avança no estado da arte ao propor um modelo para análise e predição da evolução de atributos internos de qualidade de software. Consideramos quatro características internas de sistemas de software orientado por objetos: acoplamento, coesão, hierarquia de herança e tamanho de classe. Extraímos e analisamos dados de 46 softwares de código aberto baseados em Java. Esses dados referem-se a oito métricas de software correspondentes aos atributos internos de qualidade considerados neste estudo. Definimos um novo método para analisar e prever a evolução de software. Nosso método usa análise de séries temporais, técnicas de regressão linear e testes de tendência. Neste estudo, aplicamos o método proposto para investigar como a estrutura interna de sistemas de software orientados por objetos evolui em termos dos atributos internos analisadas e neste processo identificamos os modelos que melhor explicam a evolução dos atributos internos analisadas. Além disso, investigamos como se comportam as relações entre as métricas dos atributos internos ao longo da evolução dos softwares e analisamos o conjunto de classes existente nos sistemas que afetam a

evolução desses atributos internos. Os principais resultados deste estudo revelaram dez propriedades de evolução de software, entre elas: acoplamento, coesão e herança evoluem linearmente; um percentual relevante de classes contribui para a evolução do acoplamento e do tamanho; uma pequena percentagem de classes contribui para a evolução da coesão. Os resultados também indicam que nosso método pode prever com precisão como um software evoluirá em previsões de curto e longo prazo. Em cenários reais de Engenharia de Software, os resultados desta tese de doutorado podem auxiliar os desenvolvedores no planejamento de suas estratégias para acomodar mudanças e novas funcionalidades no software de forma que a degradação da arquitetura de software possa ser mitigada ou evitada.

Palavras-chave: evolução de software, métricas de software, qualidade de software, séries temporais, análise de tendência

Abstract

Software evolution is a natural software life cycle process that consists of adapting, maintaining, and updating it. Knowing and understanding how software systems evolve can contribute to their management and quality control. This Ph.D. dissertation investigates how software systems evolve, precisely their internal quality attributes, such as cohesion and coupling. Although many studies have been done on software evolution, until now, there has not been an overview of the state of the art in modeling software evolution. Thus, our first step was to carry out a Systematic Literature Mapping (SLM) to identify what is being produced regarding models for software evolution. Our SLM identified 71 articles published from 1979 to 2022. The analysis of these studies revealed that: (i) software evolution models were built in three categories: characterization, description, and prediction; (ii) the primary purposes of models are the prediction of defects and changes, characterization of the evolution of the internal structure of the software and explanation of how the intrinsic properties of software systems evolve; (iii); researchers have invested in generating datasets to be used in software evolution modeling studies rather than using existing datasets; (iv) there are some well-known software evolution datasets publicly available in the literature, but they are not very large and are outdated; and (v) Machine learning, regression techniques, ARIMA and graph-based techniques are the main techniques used to create models. This Ph.D. dissertation advances state of the art by proposing a model for analyzing and predicting internal software quality attributes. We consider four internal characteristics of object-oriented software systems: coupling, cohesion, inheritance hierarchy, and class size. We extracted and analyzed data from 46 Java-based open-source software systems. These data refer to eight software metrics referring to the internal attributes analyzed in this study. We define a new method to analyze and predict software evolution. Our method uses time series analysis, linear regression techniques, and trend testing. In this work, we apply the proposed model to investigate how the internal structure of object-oriented software systems evolves in terms of the analyzed internal attributes. Using the proposed method, we identified the functions that best explain the evolution of the analyzed internal attributes. In addition, we investigate how the relationship between internal attributes metrics behaves along the evolution of the systems and the set of existing classes in the systems that affect the evolution of these internal attributes. The main results of this study reveal ten properties of software evolution, among them: coupling, cohesion, and inheritance evolve linearly; a relevant percentage of classes contribute to size coupling and evolution; a small percentage of

classes contribute to the evolution of cohesion. The results also indicate that our method can accurately predict how the software system will evolve in short- and long-term forecasts. In real software engineering scenarios, the results of this Ph.D. dissertation can help developers in planning their strategies to accommodate changes and new features in the software so that software architecture degradation can be mitigated or avoided.

Keywords: software evolution, software metrics, software quality, time series, trend analysis

List of Symbols

A - Abstractness
AC - Afferent Coupling
ACF - Autocorrelation
ACI - Average Complexity by Inheritance
AHF - Attribute Hiding Factor
AIF - Attribute Inheritance Factor
ARIMA - Autoregressive Integrated Moving Average
CALM - Class Aggregation Level Measure
CBO - Coupling between Objects
CCI - Complexity of Class by Inheritance
CCOM - Class Cohesion Measure
CMC - Class Method Complexity
CMCM - Class Member Complexity Measure
CICM - Class Inheritance Complexity Measure
COF - Coupling Factor
COMETS - Code Metrics Time Series
CTA - Coupling Through Abstract Data Type
CTM - Coupling Through Message Passing
DIT - Depth of Inheritance Hierarchy
DCM - Dynamic Coupling Metric
EC - Efferent Coupling
EOC - Export Object Coupling
I - Instability
ICSE - International Conference on Software Engineering
(IOC) - Import Object Coupling
LCOM - Lack of Cohesion of Methods
MHF - Method Hiding Factor
MIF - Method Inheritance Factor
MLOC - Method Lines of Code
NAC - Number of Ancestor Classes
NAP - Number of Public Attributes
NBD - Nested Block Depth
NCA - Number of Afferent Connections

NCP - Necessary Coupling
NDC - Number of Descendent Classes
NMP - Number of Public Methods
NOA - Number of Attributes
NOC - Number of Children
NLM - Number of Local Methods
NOF - Number of Fields
NOM - Number of Methods
NORM - Number of Overridden Methods
NSF - Number of Static Attributes
NSM - Number of Static Methods
PACF - Partial Autocorrelation
PAR - Number of Parameters
PMSE - Predicted Mean Square Error
PF - Polymorphism Factor
RFC - Response For Class
RMD - Normalized Distance
SFC - Strong Functional Cohesion
SIX - Specialization Index
SLM - Systematic Literature Mapping
SLOC - Source Lines of Code
SLR - Systematic Literature Review
SPL - Software Product Lines
TCC - Tight Class Cohesion
UNCP - Unnecessary Coupling
VG - McCabe Cyclomatic Complexity
WFC - Weak Functional Cohesion
WoC - World of Code
WMC - Weighted Method per Class

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Software evolution is a general term employed in Software Engineering to describe one of the phases of a software system life cycle. Usually, it defines the process of developing, maintaining, and updating software systems for various reasons [Mens et al., 2010]. These reasons may be a correction of an error and inclusion or a change in the software system requirements. Mens et al. [2010] consider software evolution and maintenance synonyms since they are related to the software life cycle and deal with similar activities, such as updating and fixing the system after its first release. Software evolution is essential because it allows including and enhancing novel features in a software system to meet the demands required by its users. However, the continuous changes generate, in most cases, an increase in the complexity of the system's internal structure, which may lead to high costs to accommodate changes and features.

Most of the total cost of a software system is due to its maintenance [Lientz and Swanson, 1980, Nosek and Palvia, 1990, Meyer, 1997, Sommerville, 2012]. It comprises from 85% to 90% of the total expenses that an organization spends with software [Erlikh, 2000, Sommerville, 2012]. The high cost and the difficulties involved in software maintenance and evolution tasks have motivated many studies.

The way software systems evolve has been a subject of research in Software Engineering for decades. As a starting point, Lehman et al. [1997] conducted empirical studies to understand the software evolution characteristics better. They describe the evolutionary nature of the software and conclude that, in general, a software system grows and undergoes maintenance continuously, has increasing complexity, and decreases quality over its evolution. They summarize their findings as eight laws, widely known as Lehman's laws. Table 1.1 presents the eight software evolution laws described by Lehman et al. [1997].

Lehman's laws are one of the landmarks of software evolution and have inspired the community to investigate this topic. The works of Lehman were done in the mid-90s, i.e., before object-orientation became popular. After, many studies investigated software evolution aiming to check and validate the presence of Lehman's laws in software development contexts, such as open-source software [Lee et al., 2007, Mens et al., 2008, Xie et al., 2009, Businge et al., 2010, Israeli and Feitelson, 2010, Alenezi and Almustafa,

Table 1.1: Lehman's laws of software evolution. Source: Adapted from Lehman et al. [1997].

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 the 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 lifetime.
V	Conservation of Familiarity	As a software system evolves, everything associated with it, such as developers, sales personnel, and users, must maintain mastery of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that 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. They must be treated as such to achieve significant improvement over any reasonable base.

2015], proprietary software [Barry et al., 2007], and mobile applications [Zhang et al., 2013, Li et al., 2017, Gezici et al., 2019].

Other works have been carried out to characterize the software evolution regarding some software internal attributes, such as size and complexity, among others [Godfrey and Tu, 2000, Capiluppi et al., 2004a,b, Capiluppi and Ramil, 2004, Robles et al., 2005, Herraiz et al., 2006, Izurieta and Bieman, 2006, Capiluppi et al., 2007, Herraiz et al., 2007a, Koch, 2007, Nasseri et al., 2008, Tempero et al., 2008, Gonzalez-Barahona et al., 2009, Hatton et al., 2017] and aiming to understand the impact of those software internal attributes on the emergence of faults, maintainability, and change [Daly et al., 1996, Cartwright, 1998, Harrison et al., 2000, Eski and Buzluca, 2011, Abuasad and Alsmadi, 2012, Couto et al., 2012, Singh and Ahmed, 2017].

The studies on the evolution of software systems' internal attributes 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 characteristic grows linearly [Robles et al., 2005], while others indicate that this growth is super-linear [Godfrey and Tu, 2000, Herraiz et al., 2006, Koch, 2007, Gonzalez-Barahona et al., 2009], sub-linear [Capiluppi and Ramil, 2004, Izurieta and Bieman, 2006, Capiluppi et al., 2007], or even follows a Pareto distribution [Herraiz et al., 2007a]. Hence, there is a gap in the current knowledge of how software systems evolve from the perspective of internal attributes. Defining methods to analyze and predict how the internal structure of the software systems evolves is important to aid software maintenance tasks.

Software evolution is a time-related phenomenon. For this reason, time series have been used in software engineering for analyzing software evolution and for defining prediction models in aspects such as defect [Couto, 2013, Raja et al., 2009, Graves et al., 2000, Arisholm and Briand, 2006, Ratzinger et al., 2007a, Wu et al., 2010], changes [Kenmei et al., 2008], clones [Antoniol et al., 2001], size [Caprio et al., 2001], complexity [Caprio et al., 2001], and quality of service (QoS) [Amin et al., 2012]. Time series consists of a collection of periods made sequentially over time [Morettin and Toloi, 2006, Bowerman and O'Connell, 1993].

1.1 Aim

This Ph.D. dissertation aims to analyze how the internal structure of software systems evolves, identify properties of software evolution, and define prediction models for software evolution in object-oriented software.

In summary, the specific aims of this Ph.D. dissertation are the following:

1. Carry out a Systematic Literature Mapping (SLM) to compile the corpus of knowledge on software evolution models.

2. Define a method for modeling and analyzing the evolution of internal attributes of software structure.

3. Investigate how the internal structure of object-oriented systems evolves from the perspective of coupling, inheritance hierarchy, cohesion, and class size.

4. Create a comprehensive software evolution dataset containing a time series of software metrics.

5. Analyze the efficiency of the proposed software evolution method for building prediction models for the following internal attributes: coupling, inheritance hierarchy, cohesion, and class size.

For the aims of this dissertation, we divided the work into two parts. In the first one, we carried out a Systematic Literature Mapping (SLM) to understand state of the art on software evolution and identify what has been produced regarding models in this topic. In the SLM, we investigate the following: (i) the types of software evolution models that researchers produced; (ii) the characteristics that have been used as the focus for the proposed models; (iii) an overview of the type of software evolution datasets existing in the literature; and (iv) the commonly used techniques to build the software evolution models.

In the second part of this Ph.D. dissertation, we defined a novel method to analyze and predict software evolution. We used it to characterize and predict object-oriented software systems' evolution from the perspective of the following internal attributes: coupling, cohesion, inheritance hierarchy, and class size. Coupling is an attribute that describes the level of dependence between the modules of a software system [Myers, 1975]. Cohesion indicates the level of relationship between the internal elements of a module [Myers, 1975]. Inheritance hierarchy is a mechanism of organizing the components within a system into a rooted tree structure so that the characteristics of a particular object may be extended by others [Tupper, 2011]. Size is a software aspect that defines a system as large or small, considering its number of lines of code, number of files, and modules [Sommerville, 2012]. In this work, we analyzed size at the class level, i.e., we investigated how the classes' sizes evolve. Other internal aspects could be investigated, but we decided to consider those four internal attributes due to their importance to software modularity since modularity is a significant factor of software quality [Meyer, 1997].

Moreover, we based our analysis on these four internal attributes because they are relevant software architecture characteristics that have been little explored and studied in the literature, except the class size, the only one that has been further investigated. Therefore, we needed more information about how coupling, inheritance hierarchy, and cohesion evolve inside the architecture of software systems.

Our proposal is based on time series. We used time series because it allows us to quantify the internal characteristics of software systems analyzed in this work by a sequence of measures that can be extracted via software metrics. We used time series with data from six software metrics to characterize these internal attributes. We considered the following software metrics: fan-in and fan-out to represent coupling; LCOM (Lack of Cohesion) and TCC (Tight Class Cohesion) to represent cohesion; NOA (Number of Attributes) and NOM (Number of Methods) to represent class size; and DIT (Depth of Inheritance Tree) and NOC (Number of Children) to characterize inheritance hierarchy.

Fan-in is the number of classes that use a given class, while fan-out is the number of classes used by a given class [Sommerville, 2012]. LCOM measures the lack of cohesion between methods of class [Chidamber and Kemerer, 1994], and TCC indicates the cohesion of a class via direct connections between visible methods [Bieman and Kang, 1995, Bär et al., 1999, Panas et al., 2005]. NOA and NOM are the numbers of attributes and methods of a class, respectively [Lorenz and Kidd, 1994]. DIT indicates a class's position in its inheritance hierarchy, and NOC is the number of immediate subclasses of a given class [Chidamber and Kemerer, 1994].

We concentrate this study on Java-based software systems for the following reasons: Java has been one of the most popular programming languages for decades, so there is a large amount of historical data from open-source Java-based software systems; there are open-source tools for collecting software metrics from Java software systems. We built a comprehensive software evolution dataset containing temporal data of open-source Java systems. Although in this Ph.D. dissertation, we considered data from six software metrics, the dataset comprises temporal information on 46 static software metrics of 46 open-source Java systems. To build it, we defined a methodology composed of four steps. (i) We carried out a selection process to choose the systems to be included in the dataset by some pre-determined requirements; (ii) we built an approach to make releases from the software systems; (iii) we collected the software metrics from the releases of the systems; and (iv) we generated the time series of the metrics values. We used the proposed dataset to carry out the modeling analysis in the Ph.D. dissertation.

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 attributes, we conducted an empirical study considering the four characteristics described in the previous paragraphs. We also 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. Based on the results of this study, we identified ten properties of object-oriented software evolution.

1.2 Contributions

The results of this Ph.D. dissertation provided the following contributions:

1. A Systematic Literature Mapping that compiles the existing knowledge on software evolution models and reveals the need for further research.

2. A novel software evolution dataset containing temporal data from 46 software met-

rics from 46 software systems. The dataset is available at `https://brunolsousa.github.io/software-evolution-dataset/index.html`

3. A novel method to analyze software evolution based on time series analysis. The technique consists of two phases. The first phase uses linear regression with a time series component to model the data's evolution pattern and identify the model type that better represents their behavior. The second one applies trend tests in the time series to analyze the classes' evolution, indicating which measures increased or decreased over time.

4. A set of ten properties that details the evolution of object-oriented software systems from the perspective of coupling, inheritance hierarchy, cohesion, and class size in a fine-grained view.

5. A forecast analysis based on the software evolution method that presented the best performance in the fitting models' phase. Short and long-term forecasts were considered.

6. A pull request into the repository of CK Took on GitHub. This pull request included new features to the tool for supporting the collection values of relevant oriented-object software metrics from Java software systems. Researchers can access details about this pull request on: <`https://github.com/mauricioaniche/ck/pull/79`>.

1.3 Publications

This Ph.D. dissertation has generated the following publications.

1. Sousa, B. L.; Ferreira, M. M.; Ferreira, K. A. M.; Bigonha, M. A. S. *Software Engineering Evolution: The History Told by ICSE.* In Proceedings of the XXXIII Brazilian Symposium on Software Engineering (SBES 2019), Salvador, BA, Brazil, pages 17–21, 2019. (Published in 2019.) Although not presented in this Ph.D. dissertation, this paper reports the results of the author's preliminary studies during his Ph.D. course. This paper concentrates on investigating how the body of knowledge of Software Engineering has evolved. In particular, we identified that software evolution is one of the main topics explored in Software Engineering.

2. Sousa, B. L.; Bigonha, M. A. S; Ferreira, K. A. M. *Analysis of Coupling Evolution on Open Source Systems.* In Proceedings of the XIII Brazilian Symposium on Soft-

ware Components, Architectures, and Reuse (SBCARS '19), Salvador, BA, Brazil, pages 23-32, 2019. (Published and **Awarded as the 2*nd* Best Paper** from the SBCARS'19.)

3. Sousa, B. L.; Bigonha, M. A. S.; Ferreira, K. A. M.; Franco, G. C. *Characterizing the Evolution of Size and Inheritance in Object-Oriented Software.* In Proceedings of the XX Brazilian Symposium on Software Quality (pp. 1-10). (Published in 2021.)

4. Sousa, B. L.; Bigonha, M. A. S.; Ferreira, K. A. M.; Franco, G. C. *A Time Series-Based Dataset of Open-Source Software Evolution.* In Proceedings of the 19th International Conference on Mining Software Repositories (pp. 702-706). (Published in 2022.)

5. Sousa, B. L.; Bigonha, M. A. S; Ferreira, K. A. M.; Franco G. C. *Evolution of Internal Dimensions in Object-Oriented Software - A Time Series Based Approach.* Submitted to Software Practice & Experience Journal, pages 1–38, 2022. (Under the second stage of the review process.)

6. Sousa, B. L.; Bigonha, M. A. S; Ferreira, K. A. M.; Franco G. C.. *Models on Software Evolution - A Systematic Literature Mapping.* Submitted to an international journal, pages 1–17, 2023. (Under review.)

1.4 Overview

This section presents an overview of our studies in this Ph.D. dissertation. Figure 1.1 shows the sequence scheme in which we carried out the studies.

Initially, we conducted a Systematic Literature Mapping (SLM) to summarize the knowledge on software evolution models and understand the need for further research on this topic. The SLM revealed gaps in the area, such as a lack of large and updated software evolution datasets and a lack of models that help developers project the evolution of the internal attributes of the software.

The second part of this work focused on covering one of the gaps found by the SLM. We created a large and updated software evolution dataset based on Java software systems. We focused on systems developed in Java since it is one of the most used programming languages by researchers and practitioners. We used GitHub to collect the data and construct our dataset. As a result, we generated a temporal time series dataset regarding 46 different software metrics from 46 software systems.

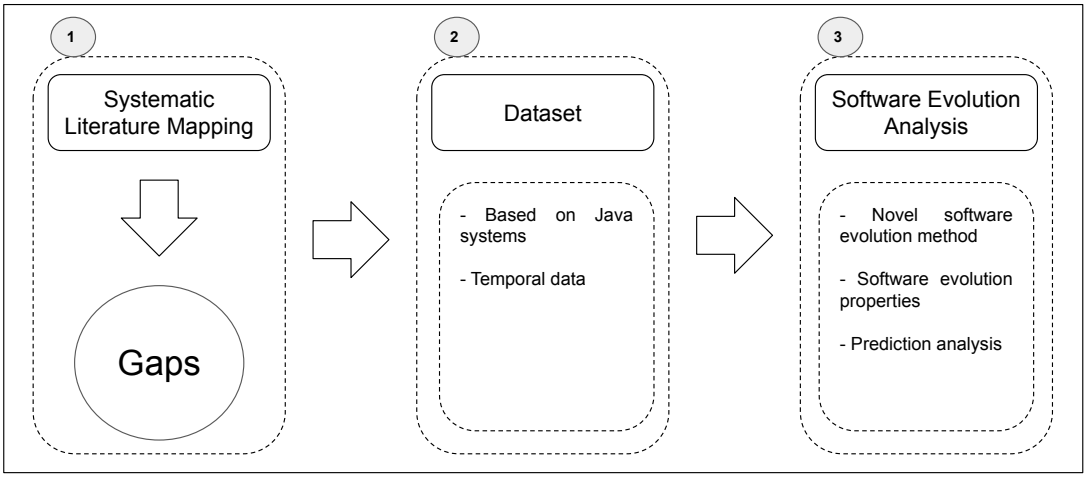

Figure 1.1: Sequence of the studies carried out inside this Ph.D. Dissertation.

The third part of this work focused on covering another gap found by our SLM. We identified the lack of models for helping developers analyze and project the evolution of internal software attributes of software systems. Then, as the first step, we defined a time series-based method composed of two phases to model temporal data of internal quality attributes. As the second step, we applied our method for analyzing the evolution of four software internal attributes: (i) cohesion, (ii) coupling, (iii) inheritance hierarchy, and (iv) size. After these analyses, we summarized all observations and identified ten software evolution properties. This part of this Ph.D. dissertation also performed a prediction analysis to evaluate our method and show its accuracy for building prediction models.

1.5 Organization of the Dissertation

We organized the remainder of this Ph.D. dissertation as follows.

Chapter 2 describes the main concepts that support this work, such as software quality, object-oriented software metrics, and time series. Besides, we define and distinguish the concepts of measurement, metrics, and measure. We present and detail the leading and most known suites of object-oriented software metrics in the literature.

Chapter 3 describes the Systematic Literature Mapping (SLM) on software evolution models we carried out to compile the knowledge existing in the literature about this topic.

Chapter 4 presents the construction of a comprehensive software evolution dataset we used to carry out our empirical studies and extract the software evolution properties.

Chapter 5 presents and describes the novel method for software evolution analysis

based on time series.

Chapter 6 reports the main observations and results we found about the evolution of coupling, cohesion, inheritance hierarchy, and class size in object-oriented software systems. We also conduct an analysis to assess the efficiency of the proposed method (Chapter 5) in extracting prediction models. Besides, this chapter summarizes the evolution properties we extracted, considering the observations found during the research questions' answers.

Chapter 7 concludes this thesis and presents some proposals for future works, and Appendix A presents the results obtained for RQ1 after applying the first phase of our time series-based method for modeling the global time series of the internal attributes software metrics.

Chapter 2

Background

This chapter builds a background for supporting the concepts used in this Ph.D. dissertation. Section 2.1 describes object-oriented software metrics and contextualizes the ones existing in the literature to measure software systems' internal characteristics. Section 2.2 contextualizes time series by presenting its main concepts and examples of application inside Software Engineering. It is important to highlight that although we discussed many software metrics in this section, we do not use all of them in the empirical analysis carried out in this Ph.D. dissertation. We chose only some of them, the ones able to measure and represent the analyzed internal attributes focused on modularity for conducting our empirical studies.

2.1 Object-Oriented Software Metrics

In internal software structure, metrics evaluate software internal attributes such as modularity, complexity, size, coupling, and cohesion. We carried out a study that analyzed the main topics investigated by works published in the International Conference on Software Engineering (ICSE). A conclusion of this study is that software metrics are among the ten most explored topics of the area [Sousa et al., 2019b]. Since the 1990s, the literature has provided many metrics to support the software's internal structure analysis. Among the object-oriented software metrics proposed so far, the ones that stand out due to their high use are proposed by Chidamber and Kemerer [1994]. They are widely known as CK metrics, representing a landmark of this field of knowledge.

Nevertheless, besides CK, several other object-oriented software metrics are proposed in the literature. As we based our study on these metrics, we surveyed the main ones described in the literature. Sections 2.1.1 and 2.1.2 present the CK and the MOOD metrics, respectively. Sections 2.1.3 and 2.1.4 describe Martin's metrics [Martin, 1994] and some complexity indicators. Section 2.1.5 discusses the set of classes proposed by Lorenz and Kidd [1994]. We present in Sections 2.1.6, 2.1.7, and 2.1.8 the group of met-

rics proposed by Li [1999], Malik and Chhillar [2011], and Mishra [2012] in this sequence. Finally, we conclude our discussion about object-oriented software metrics by presenting and detailing some dynamic metrics suites in Section 2.1.9.

2.1.1 CK Metrics

The CK metrics consist of object-oriented metrics proposed by Chidamber and Kemerer [1994], aiming to assess the following software aspects: coupling, inheritance hierarchy, and cohesion. This group is composed of six metrics. They are Weighted Method per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling between Objects (CBO), Response For Class (RFC), and Lack of Cohesion of Methods (LCOM). We detail each one of these metrics as follows.

- **WMC (Weighted Methods per Class)** measures the complexity of a given class by considering the sum of all complexity of the methods that compose it. To compute WMC, weights for each class method must be assigned, which a secondary metric may infer. Chidamber and Kemerer [1994] do not define a specific complexity indicator for use together with this metric, and then, they may determine weights via code lines and cyclomatic complexity of each method. According to the authors, WMC indicates efforts to develop and maintain a respective class. Therefore, the higher the number of methods in a class, the higher the tendency of that class to be less specific, limiting its reuse and harming its cohesion.

- **DIT (Depth of Inheritance)** is related to the inheritance hierarchy in object-oriented software. It represents the level at which we positioned a given class in an inheritance hierarchy within an object-oriented software system, i.e., the distance between its current position and the root of its inheritance tree. Then, the farther a class is from the root of the inheritance tree, the higher the value of these metrics. According to Chidamber and Kemerer [1994], intense inheritance hierarchy trees indicate complex structures and, consequently, impair the comprehension of the module and make it more prone to error.

- **NOC (Number of Children)** also refers to the inheritance hierarchy and indicates the number of immediate subclasses a particular class has. Chidamber and Kemerer [1994] highlight that the higher the value of this metric for a class, the higher its reuse level. Besides, components with high values of NOC require more attention tests to avoid errors introduced into the superclass propagating over its subclasses.

- **LCOM (Lack of Cohesion of Methods)** is related to cohesion in object-oriented software. It uses the notion of the similarity degree of methods to measure the level of cohesion of a class and how complex it has been designed. According to Chidamber and Kemerer [1994], methods that access one or more attributes inside their respective class are considered similar methods. Then, they defined the calculation of LCOM as the difference between the similar and non-similar methods in a class. In this way, if all possible pairs of methods existing in the class share the attributes, the value of LCOM will be 0.

  On the other hand, if no pair of methods have common attributes, the LCOM of that class will be 1. Therefore, the range values of this metric vary from 0 to 1. The higher its value in a class, the less cohesion degree of this respective class.

- **CBO (Coupling Between Objects)** measures the number of classes it calls through an association relationship. The association between two classes may occur when one accesses a variable or method defined in the other. This metric reflects the coupling of a component and establishes its degree of dependency inside a software system. Chidamber and Kemerer [1994] indicate that classes with a high coupling level have low potential reuse and are more prone to change when other system parts are modified.

- **RFC (Response for a Class)** indicates the number of methods that may execute in response to a message received by an object of the class. RFC gives its result considering the class methods sum and the set of methods triggered by each method belonging to the analyzed class. Chidamber and Kemerer [1994] point out that the higher the RFC in a class, the more complex and challenging it will be to test and maintain it. Like CBO, RFC also indicates the coupling of a class.

2.1.2 MOOD Metrics

Abreu and Carapuça [1994] proposed the MOOD metrics to measure the five aspects of object-oriented software: inheritance hierarchy, encapsulation, coupling, polymorphism, and software reuse. An essential element of the MOOD metrics is that the ratio always computes them. The numerator reflects the quantity of a particular aspect extracted in the software, and the denominator represents the maximum possible value of that aspect. Therefore, this group of metrics will always vary between 0 and 1. The following six metrics compose this group.

- **MIF (Method Inheritance Factor)** consists of the ratio between the sum of the number of inherited methods in all system classes by the total number of methods existing. Its result indicates the use of inheritance in the systems, and a high value for this metric implies that the system has a high level of reuse.

- **AIF (Attribute Inheritance Factor)** is similar to MIF but considers attributes instead of methods. To compute AIF, we have to divide the number of inherited attributes of the system by the total number of attributes existing. As well as to MIF, AIF values close to 1 indicate a high degree of data reuse.

- **COF (Coupling Factor)** evaluates the coupling of the system as a whole. Abreu and Carapuça [1994] consider this metric as a client-server relationship, where one class is responsible for providing services, and the other type is responsible for consuming services. To compute COF, we have to divide the number of real connections in the systems, considering the sum of the class connections, by the largest possible number of connections for the software. When a software system is completely connected, the value of this metric is 1.

- **PF (Polymorphism Factor)** defines the ratio between the number of polymorphism cases identified in the system and the maximum number of possible polymorphism cases in the system. Values close to 1 indicate a great use of polymorphism, while values close to 0 show the system has a low resource consumption.

- **MHF (Method Hiding Factor)** indicates the percentage of methods hidden in the system. It is a metric computed to the system level that divides the number of hidden methods by the total number of methods from the systems. Abreu and Carapuça [1994] indicate that this metric reflects the level of encapsulation applied in a software system, i.e., how hidden the implementation details are. The higher the number of methods hidden, the closer this metric will be to 1, while the closer to 0 the value of this metric, the more the number of public methods to the system's users. Such a scenario indicates a low degree of encapsulation of the system.

- **AHF (Attribute Hiding Factor)** is similar to MHF but considers hidden attributes rather than hidden methods. We compute this metric by the ratio between the number of hidden attributes in all system classes and the total number of attributes defined in the system. AHF values close to 1 show that the system is well hidden concerning its internal data. On the other hand, values close to 0 indicate that the system has a high quantity of public attributes. Ferreira [2006] suggests that attribute hiding is essential to ensure the independence of classes in object-oriented software and force the system to establish a communication channel based on the class interface and not on their internal implementation. Therefore, the ideal

scenario for software quality is that systems have values equal to or as close to 0 as possible.

2.1.3 Martin's Metrics

This section brings up the metrics proposed by Martin [1994]. Martin [1994] defined a class category concept to present these metrics. A class category is a cohesive group of classes. If one class in this category is changed, the other classes also have a high chance of being changed; classes are reused together, and classes have a common goal or perform interdependent functions. This group is composed of five metrics. They are described in the sequel.

- **AC (Afferent Coupling)** indicates the number of external classes that consume the services of the classes belonging to a particular category. According to Martin [1994], the higher the value of AC, the higher the coupling level.
- **EC (Efferent Coupling)** measures the number of internal classes in a given category that depends on external classes in this category. According to Martin [1994], this metric is another coupling indicator. The higher its value, the higher the level of coupling in the group.
- **I (Instability)** depends on the value of afferent and efferent coupling to be extracted. Its computation appears in Equation 2.1.

$$I = \frac{EC}{AC + EC} \tag{2.1}$$

  According to Martin [1994], this metric ranges from 0 to 1. The values of I close to 1 indicate that the analyzed category of classes is more unstable, while close to 0 shows that the analyzed category of classes is more stable.

- **A (Abstractness)** is defined as the ratio between the number of abstract classes within a category and the total number of classes in this category.
- **RMD (Normalized Distance)** measures the distance between instability (I) and abstraction (A) by using Equation 2.2.

$$RMD = |A + I + 1| \tag{2.2}$$

Martin [1994] defined a region of balance between instability (I) and abstraction (A) that he labeled as the Main Sequence. Then, by extracting RMD, we may measure how far the relationship between these two concepts is from the Main Sequence region. The higher this distance, the lower the balance between instability and abstraction.

2.1.4 Complexity Metrics

This section presents six metrics often used in the literature as indicators of software complexity.

- **Fan-in** measures the number of classes that reference a particular class. For instance, given a class X, the fan-in of X would be the number of classes that call X by referencing it as an attribute, accessing some of its attributes, or invocating some of its methods. Fan-in is a metric at the class level. According to Sommerville [2012], high fan-in values mean the class is strongly coupled to the remainder of the project. Then, changes in this class will impact extensive repercussions and changes in other parts of the program.

- **Fan-out** is the number of other classes referenced by a particular class. In other words, given a class X, the fan-out of X is the number of classes called by X via attributes reference, method invocations, or object instances. As well as fan-in, fan-out is also a metric at the class level. According to Sommerville [2012], a high value for this metric suggests high complexity for the analyzed component arising from the complexity of the control logic necessary to coordinate the called elements.

- **MLOC (Method Lines of Code)** defines the number of lines of code existing in a method from a particular class.

- **SIX (Specialization Index)** aims to evaluate how much a particular class overwrites the superclass behavior [Lorenz and Kidd, 1994]. SIX is a secondary metric that requires three other primary metrics for its computation. We extract SIX by the ratio between the number of overwritten methods (NORM), weighted by the level of the class in an inheritance hierarchy (DIT), and the total number of methods (NOM). Equation 2.3 shows how to compute SIX in a given class.

$$SIX = \frac{NORM \times DIT}{NOM} \tag{2.3}$$

- **NBD (Nested Block Depth)** measures the depth of nested blocks in a method. Nested blocks occur when control structures, like conditional (*if*) and repetition loops (*for* and *while*), are inserted inside each other. NBD indicates complexity since the increased nested block in the source code makes it harder to understand.

- **VG (McCabe Cyclomatic Complexity)** evaluates the complexity of methods in object-oriented software [McCabe, 1976]. VG aims to measure the number of independent execution paths in source code. For this purpose, a graph models the source code's execution flow, where the nodes consist of the command blocks, and the directed edges indicate the execution flow from one block to another. For instance, suppose that there is a source code with two command blocks, A and B, and its execution flow goes from A to B. Modeling this code as a graph, we would have two nodes, A and B, and a directed edge that connects them in the direction from A to B. After modeling the source code by a graph, the computation of the VG metric appears in Equation 2.4.

$$VG = N - C - E \tag{2.4}$$

  In equation 2.4, N consists of the number of nodes, E indicates the number of edges, and C is the number of connected components in the graph.

- **TCC (Tight Class Cohesion)** measures the cohesion of a class by the degree of connectivity between visible methods in a class [Bieman and Kang, 1995, Bär et al., 1999, Panas et al., 2005]. It produces a value between 0 and 1. Then, the higher its value in a class, i.e., closer to 1, the more cohesion degree of this respective class.

2.1.5 Lorenz and Kidd's Metrics

This section brings up the set of metrics defined by Lorenz and Kidd [1994]. This group of metrics aims to evaluate some static aspects of a software system, e.g., inheritance hierarchy, size, and the classes' internal properties. A total of ten metrics compose it, and all of them, except PAR, are at the class level. We present and discuss each of them, except SIX already described in Section 2.1.4, as follows:

- **NCA (Number of Afferent Connections)** refers to the coupling aspect; it measures the number of classes using a particular class's services.

- **NMP (Number of Public Methods)** computes the number of public methods that a particular class has. With NMP, we can characterize the size of a class and the number of services it provides.
- **NAP (Number of Public Attributes)** measures the number of public attributes. As well as NMP, NAP expresses the size aspect of the software.
- **NOA (Number of Attributes)** computes the total number of attributes belonging to a particular class. NOA considers the public, private, and protected attributes at the moment of its calculation. The literature also refers to this metric as the number of fields (NOF).
- **NOM (Number of Methods)** computes the total number of methods belonging to a class. NOM considers public, private, and protected attributes in its count.
- **NORM (Number of Overridden Methods)** refers to the inheritance hierarchy. It measures the number of methods of a particular class overwritten by its subclasses.
- **NSF (Number of Static Attributes)** computes the number of attributes declared as static in the classes.
- **NSM (Number of Static Methods)** computes the number of methods declared static in the classes.
- **PAR (Number of Parameters)** measures the total number of parameters for each method of the analyzed project.

2.1.6 LI's Metrics

According to Li [1999], the set of metrics proposed by Chidamber and Kemerer [1994] has shortcomings and does not cover some relevant aspects of the software during the measurement process. Li [1999] proposed six object-oriented software metrics to fill the gaps left by the CK metrics. The metrics defined by Li [1999] evaluate the inheritance hierarchy, size, complexity, and coupling aspect in the software. We present and describe each of them as follows:

- **NAC (Number of Ancestor Classes)** DIT inspired this metric. As described in Section 2.1.1, DIT aims to identify the number of classes that influence a particular type by computing the distance of that component to the root of the inheritance tree. Li [1999] argues that in software developed in a programming language that

does not support multiple inheritances, e.g., Java, DIT provides efficient measures. However, the same does not occur in software built-in languages that offer this resource, e.g., C++, since DIT could not discern if, at a particular level, a class inherits from one or more classes. Then, DIT may not return the correct quantity of ancestral types in this situation. Due to this, Li [1999] proposed NAC to measure the number of classes that precede a particular class in an inheritance hierarchy, not only the number of levels. With this metric, the author believes that supplied this deficiency of DIT for software developed with the support of multiple inheritances.

- **NDC (Number of Descendent Classes)** is similar to NOC, but with a bit of change in its purpose. NOC measures the number of immediate subclasses of a particular class in an inheritance hierarchy. Then, considering that class A has class B as its direct child and class B has many subclasses as successors in the inheritance tree, the children of B are not considered in the NOC value of class A. The NDC metric also measures the number of descendants of a given class. However, it analyzes all posterior levels of the diagnosed type in the inheritance hierarchy and not only its immediate posterior level.

- **NLM (Number of Local Methods)** captures the number of methods belonging to a class accessible to others, e.g., public methods. According to Li [1999], NLM refers to the size aspect. However, the high values of this metric may indicate that the software has a low cohesion and a high degree of coupling.

- **CMC (Class Method Complexity)** aims to summarize the internal complexity of all methods existing in the class. This metric concept is very similar to the WMC but differs in how they express complexity in the methods. While the classic implementation of WMC attributes the value 1 to each method in a class and makes WMC equal to the number of methods (NOM), CMC attributes each method's complexity as its lines of code (MLOC). According to Li [1999], CMC provides a better view of a class's development and maintenance costs than WMC.

- **CTA (Coupling Through Abstract Data Type)** determines the coupling level of a particular class in terms of data. According to this metric, class A has a data coupling with class B when A uses B to define its attributes. Class A is coupled through abstract data type with B because it instantiates B as one of its objects. Therefore, CTA reflects the total number of classes defined as attributes in a particular class.

- **CTM (Coupling Through Message Passing)** aims to provide a view of the coupling in a class regarding the level of services it consumes from another. With this metric, class A has a service coupling with class B when A invokes one or more

methods defined in B. Therefore, CTM indicates the total of messages a particular class shares with other types through the method call mechanism.

2.1.7 Malik & Chhillar's Metrics

Malik and Chhillar [2011] proposed four software metrics at the class level to assess the object-oriented software quality. These metrics express the complexity, coupling, and cohesion aspects.

- **CMCM (Class Member Complexity Measure)** reflects the sum of the total number of attributes and methods public and protected in a particular class. According to Malik and Chhillar [2011], the high values of this metric indicate that the component has a low encapsulation level.
- **CICM (Class Inheritance Complexity Measure)** considers that C is a class in the system, and A_i, such as $1 \leq i \leq n$, consists of the set of parents classes of C. Then, we extract CICM in the following way:

$$CICM(C) = n + \sum_{i=1}^{n} CICM(A_i) \tag{2.5}$$

- **CALM (Class Aggregation Level Measure)** measures the level of coupling in the class. It is expressed by the ratio between the number of attributes defined as types in other classes and the total number of attributes defined in the classes. The higher the value of this metric, the more coupled the class is.
- **CCOM (Class Cohesion measure)** evaluates cohesion and is very similar to the LCOM. Equation 2.6 exhibits the computation of CCOM.

$$CCOM = \frac{N_1 + N_2 + ... + N_m}{m \times (n-1)} \tag{2.6}$$

In Equation 2.6, $N_i = N_1 + N_2 + ... + N_m$ consists of the total number of methods that use the same attribute in the class, m consists of the total number of attributes, and n is the total number of methods in the class.

2.1.8 Mishra's Metrics

Mishra [2012] proposed two metrics for evaluating the inheritance hierarchy in object-oriented software systems. We present and detail them as follows.

- **CCI (Complexity of Class by Inheritance)** expresses the complexity of a particular class by taking into account the complexity of its antecedents in an inheritance hierarchy. Mishra [2012] defined the current metric, considering that when we include a class in an inheritance hierarchy with other entities, it takes on the characteristics of its antecedents. To compute this metric, consider that C is a class in a system, and M_i, such as $1 \leq i \leq n$, consists of the set of methods in this class. Besides, consider that C is a subclass of the set of superclasses denominated by A_j, such as $1 \leq j \leq m$. Then, to compute CCI(C), we would have to apply Equation 2.7:

$$CCI(C) = \sum_{i=1}^{n} complexity(M_i) + \sum_{j=1}^{m} CCI(A_j) \tag{2.7}$$

  CCI is the sum of the complexity of the methods existing in the analyzed class plus the sum of the CCI values of all its superclasses. According to Mishra [2012], high values for this metric indicate class is more complex and prone to fail.

- **ACI (Average Complexity by Inheritance)** consists of an arithmetic average of the CCI values regarding all software classes. It is measured at the system level and provides a global view of the complexity level of the software systems.

2.1.9 Dynamic Metrics

The metrics presented in the previous section are static. Static metrics characterize software properties, such as coupling and cohesion, by analyzing the software's static aspects, e.g., source code. However, they may not assess the dynamic behavior of an application at runtime since the execution environment influences it. Because of this, another category of software metrics, labeled as dynamic metrics, was created to solve this problem.

Dynamic metrics consist of a category of software metrics that consider the execution traces of the software code or its executable models to capture the dynamic behavior

of the software system [Chhabra and Gupta, 2010]. For a better understanding of the difference between dynamic and static metrics, Table 2.1 compares these two metrics categories.

Table 2.1: Comparison between static and dynamic metrics. Source: Chhabra and Gupta [2010]

Static Metrics	Dynamic Metrics
Simple to collect	Difficult to obtain
Available at the early stages of software development	Accessible very late in the software development lifecycle
Less accurate than dynamic metrics in measuring qualitative attributes of software	Suitable for measuring quantitative as well as qualitative attributes of software
Deal with the structural aspects of the software system	Deal with the behavioral aspects of the system also
Inefficient for dealing with dead code and OO features such as inheritance, polymorphism, and dynamic binding	Dynamic metrics are capable of dealing with all object-oriented features and dead code
Less precise than dynamic metrics for the real-life systems	More accurate than static metrics for real-life systems

This section enlists and discusses the leading and most relevant suites of dynamic metrics in the literature. For better comprehension, we describe the dynamic metrics suites according to their analyzed properties in the sequel.

Coupling. We describe here the dynamic metrics suites proposed to measure the coupling.

- **Yacoub's Metrics** refer to two dynamic metrics, Export Object Coupling (EOC) and Import Object Coupling (IOC) [Yacoub et al., 1999]. These metrics measure the coupling property in object-oriented software systems. The main goal of these metrics is to express the intensity of the interactions between two objects at the runtime in a given scenario, i.e., during the execution of a software feature. Then, considering two objects, o_i and o_j, in a scenario x, the $EOC_x(o_i, o_j)$ is computed by dividing the number of messages sent from o_i to o_j by the total number of messages exchanged during the execution of scenario x. Similarly, the $IOC_x(o_i, o_j$ is extracted by dividing the number of messages that o_i received from o_j by the total number of messages exchanged during the execution of scenario x.

- **Arisholm's Metrics** extend the concepts of import and export couplings defined by Yacoub et al. [1999]. Besides that, Arisholm et al. [2004] propose a suite of 12 different dynamic coupling metrics. They consider three orthogonal dimensions: direction, mapping, and strength, to define these metrics, and each dimension establishes a different characteristic in their concept.

Arisholm et al. [2004] defined their nomenclature considering the orthogonal dimensions to facilitate understanding the concept of these metrics. Each dynamic metric starts with *EC* or *IC* to indicate its direction and discern between *import coupling* and *export coupling*. In this suite, *import coupling* means that the metric will measure the messages sent from an object or class, whereas *export coupling* will compute the messages received by an object or class. The following letter in the metrics nomenclature indicates the mapping of the metric. It may be *O*, which designates that the metric refers to an object, or *C*, which specifies that it refers to a class. The last letter in the metric name associates its strength, which may assume three possibilities: *D (Dynamic messages)*, *M (Distinct method invocations)*, and *C (Distinct classes)*. The strength of the metric defined as *D* expresses that the object or class is sending or receiving a dynamic message. The *M* strength shows that the metric is computing invocations of methods, and the *C* indicates that the metric is measuring the use of a class.

An example of a dynamic metric proposed by Arisholm et al. [2004] is *IC_OC*. By the name of this metric, it computes the number of distinct server classes used by the methods of a particular object.

- **Mitchell's Metrics** evaluate the coupling between objects at different levels. However, according to Mitchell and Power [2005, 2006], these metrics defined by Arisholm et al. [2004] do not measure the degree of the coupling. Then, Mitchell and Power [2005, 2006] tried to fill this gap by proposing a suite composed of seven dynamic metrics, of which three were based on the CBO metric and aimed to measure the class coupling level at the runtime. The other four metrics aim to characterize the dynamic coupling at the object level.

- **DCM (Dynamic Coupling Metric)** measure the influence of one object on others over some time. Hassoun et al. [2004a,b, 2005] proposed this metric. It is essential to highlight that the authors defined this metric for assessing software built on declarative control languages that allow the writing of specifications of the program behavior. It is necessary to extract the DCM metric to observe the history of the object, i.e., the sequence of its states in time. Then, during the period provided before collecting this metric, DCM is extracted by the sum over all program execution steps and the sum of the total number of objects coupled to the analyzed object.

Cohesion. We describe here the dynamic metrics suites proposed in the literature to measure the cohesion.

- **Gupta's Metrics** involve two dynamic cohesion metrics: Strong Functional Cohesion (SFC) and Weak Functional Cohesion (WFC). Gupta and Rao [2001] based on

dynamic slicing to propose these metrics and consider both the definition and uses of the class attributes in the methods instead of only use as occurring in the static cohesion metrics. Dynamic slicing is an approach that computes the set of statements, the program slice, whose execution may affect the value of a given variable at some point of interest [Weiser, 1984]. The authors define SFC as that measure arising out of def-use pairs of each common type to the dynamic slices of all the output variables. Similarly, the WFC consists of the measure arising out of def-use pairs of each type found in dynamic portions of two or more output variables.

- **Mitchell's Metrics** refer to Runtime Simple LCOM (R_{LCOM}) and Runtime Call-Weighted LCOM (RW_{LCOM}) dynamic cohesion metrics. Mitchell and Power [2003, 2004] based on the concept of LCOM to propose these two metrics. R_{LCOM} computes the cohesion in a class in the same way as the static LCOM. Still, it uses the variable instances that have been accessed at runtime instead of considering the pairs of methods that use common variables in their source code. RW_{LCOM} is an extension of R_{LCOM}, and it weights each accessed variable instance by the number of times it is accessed at runtime.

Complexity. In this section, we describe the dynamic metrics suites proposed in the literature to measure complexity.

- **Munson's Metrics** measures the complexity of a particular component by extracting the product of its static relative complexity and its probability of execution. Khoshgoftaar et al. [1993], Munson and Khoshgoftaar [1996] defined these dynamic metrics. The relative complexity of a module consists of classifying the components' various complexity metrics in a few independent complexity domains and then mapping these domains to a single metric Munson and Khoshgoftaar [1992]. The probability of execution of an element implies the actual traces of execution of the software obtained using profiling tools.

- **Yacoub's Metrics** measure the operational complexity of objects. Yacoub et al. [1999] proposed this dynamic complexity metric. It uses the McCabe Cyclomatic Complexity (VG), which is computed at the object level. To extract it, we have to identify all possible scenarios of the software, i.e., the states that it may assume at runtime, and compute the cyclomatic complexity for the analyzed object in each scenario and the probability of each scenario being executed. We summarize the final value of the dynamic complexity metric proposed by Yacoub et al. [1999] by the sum of the products between the cyclomatic complexity of an object in each scenario and the probability of that scenario occurring at software runtime. Equation 2.8 illustrates the computation of this metric.

$$OCPX(o_i) = \sum_{x=1}^{|X|} PS_x \times ocpx_x(o_i) \tag{2.8}$$

In Equation 2.8, $|X|$ is the total of scenarios that the software may assume, x is a specific scenario, PS_x consists of the probability of a given scenario x_i occurs at runtime, $ocpx_x(o_i)$ indicates the cyclomatic complexity of an object in the scenario x_i.

2.2 Time Series

In statistics and econometrics, a time series consists of a collection of periods made sequentially over time [Morettin and Toloi, 2006, Bowerman and O'Connell, 1993]. In general, the periods in a time series are serially correlated. When working with this kind of data, the main concern is identifying the pattern that better describes their behavior. Time series are often used in several areas, such as economy, meteorology, and medicine, to describe the phenomenon of interest over time and build forecasts for future values to support decision-making in certain situations [Morettin and Toloi, 2006].

Formally, we write a time series (Z) as $Z_t = \{z_1, z_2, ..., z_t\}$, where t indicates the size of the time series [Morettin and Toloi, 2006]. According to Morettin and Toloi [2006], time series may be discrete and continuous. A discrete time series consists of periods made in fixed time intervals, i.e., period intervals that are usually equally spaced out. For instance, when daily or monthly measured the temperature is considered a discrete time series. However, continuous time series is when the periods are continuously taken in the time, i.e., the period time belongs to the real numbers.

In a time series, the period intervals are generally desired to be equally spaced out over the total period. In cases where the time series interval is not equally spaced, the final analysis may lead to erroneous conclusions if specific forms of modeling that deal with this problem are not applied.

In Software Engineering, there are works in which time series are applied, e.g., time series of software metrics to support defect prediction Couto et al. [2014] and to analyze the internal structures of software over the evolution process [Herraiz et al., 2006, Koch, 2007, Israeli and Feitelson, 2010].

2.3 Final Remarks

Time series consists of a collection of periods made sequentially over time. In this work, we applied a time series of software metrics to analyze how object-oriented software systems evolve in coupling, cohesion, size, and inheritance.

Many software metrics are proposed in the literature, especially metrics for measuring object-oriented software. Among the several suites available, the most known and used are the CK metrics [Chidamber and Kemerer, 1994]. However, the CK metrics do not contain all the metrics related to the aspects considered in this Ph.D. dissertation. Given the large quantity of object-oriented software metrics available in the literature, we chose a set of software metrics among the ones discussed in this section, for carrying out the empirical analysis in this work. We defined two software metrics for each internal attribute analyzed since they allow us to study the evolution of different characteristics inside the same attribute. Then, we chose the software metrics LCOM and TCC for cohesion, fan-in and fan-out for coupling, DIT and NOC for inheritance hierarchy, and NOA and NOM for size.

Aiming to provide an overview of the state-of-the-art on models for software evolution, Chapter 3 presents a Systematic Literature Mapping (SLM). The SLM details the main lines of research on this topic by showing the main findings of the studies and the gaps that previous researchers have not yet covered.

Chapter 3

Systematic Literature Mapping on Software Evolution Models

To comprehend how software systems evolve and support the decisions to improve software structure, the community has invested in building models to represent the evolution behavior of a given aspect of software systems or to identify events that may indicate future problems. Many works have considered internal software properties, such as size and coupling, to estimate the probability that a system will present defects over its lifetime [Graves et al., 2000, Arisholm and Briand, 2006, Aversano et al., 2007, Ratzinger et al., 2007a, Shatnawi and Li, 2008, English et al., 2009, Raja et al., 2009, Wu et al., 2010, Yang et al., 2014, Zong, 2020, Walunj et al., 2022]. Other works have aimed to model the internal structure of the software systems to understand their behavior and identify patterns they tend to follow [Woodside, 1979, Godfrey and Tu, 2000, Lehman et al., 2001, Ohlsson et al., 2001, Herraiz et al., 2005, Robles et al., 2005, Herraiz et al., 2007a, Koch, 2005, Leskovec et al., 2005, Koch, 2007, Kirbas et al., 2014, Ruohonen et al., 2015, Grbac and Mauša, 2016, Sousa et al., 2019a, 2021]. Those studies show that software evolution models are essential to support developers and researchers in making good decisions to improve software quality and prevent problems.

Considering the importance of software evolution models, it is necessary to compile the content existing in the literature, aiming to provide an overview of this topic and identify its main gaps. There are some efforts in this direction. For instance, Breivold et al. [2010], Syeed et al. [2013], and Rasool and Fazal [2017] carried out a systematic literature review (SLR) on software evolution concentrated in open source software systems. In contrast, Marques et al. [2019] based their SLR on software product lines, and Vogel-Heuser et al. [2015] focused on automated production systems. Although these efforts brought interesting insights, they did not investigate software evolution models.

This chapter presents an SLM aiming to investigate the state-of-the-art on software evolution models. More specifically, this SLM aims to identify: (i) the main types of models that the researchers have built; (ii) the purpose for which the software evolution models have been built; (iii) the characteristics of the datasets that have been used to build the software evolution models; and (iv) the techniques that have been used to create

the models.

We analyzed 71 primary studies. The main results of this SLM show that:

- There are three types of models for software evolution: Characterization, Descriptive, and Prediction.
- Prediction models concentrate mainly on predicting defects and changes. Characterization models consider the evolution of the internal software structure. Descriptive models explain how the intrinsic properties of software systems evolve.
- The public software evolution datasets proposed in the literature are deficient. The existing ones are not too large and are outdated. Due to these facts, researchers have created their dataset for carrying out studies. However, those datasets also are outdated in some cases and are not too large in most cases.
- There is a high diversity of approaches for building software evolution models. The ones that have stood out: Regression techniques, ARIMA, and Graph-based techniques.

We organize the remainder of this chapter as follows. Section 3.1 describes the planning and the execution of this SLM and describes the search and the selection of primary studies. Section 3.2 presents the main findings of this SLM and Section 3.3 discusses the results. Section 3.4 shows the main threats to the validity of this study and discusses the main decisions we have taken to mitigate them. Section 3.5 presents the related works. Section 3.6 concludes this study this chapter by highlighting the main findings and contributions provided by this SLM.

3.1 Systematic Literature Mapping Protocol

Systematic Literature Mapping (SLM) and Systematic Literature Review (SLR) are two important ways to aggregate and build knowledge in a specific area. For Kitchenham and Charters [2007], Systematic Literature Mapping is “a broad review of primary studies on a specific subject that seeks to identify the available evidence on a particular topic”. In contrast, a Systematic Literature Review is defined as “a means of identifying, evaluating, and interpreting all available evidence relevant to a specific issue, thematic area, or phenomenon of interest”. This chapter presents an SLM since we conducted a broad review of software evolution models. The goals of an SLM are: (i) to provide a wide overview of a research area; (ii) to establish if research evidence exists on a topic;

(iii) to identify gaps that have not yet been covered by studies in a particular area; and (iv) to the quantity of evidence Kitchenham and Charters [2007].

The SLM presented in this paper addresses studies regarding models of software evolution. We conducted this SLM in three phases: planning, execution, and analysis.

3.1.1 Planning

The planning phase describes the protocol used for conducting the SLM. The activities carried out in this phase are: (i) definition of the research questions; (ii) selection of the databases to search the primary studies; (iii) construction of the search string; and (iv) application of the inclusion and exclusion criteria.

Research Questions. The research questions (RQ) aim to investigate the state-of-the-art of software evolution models and understand how the researchers have stated this topic in the literature.

We defined four research questions to be investigated in this study.

RQ1: What kinds of models on software evolution are proposed in the literature?

RQ2: Which software aspects have been considered in software evolution models?

RQ3: What are the main features of the datasets used in studies on software evolution?

RQ4: What approaches have been applied for building models on software evolution?

Electronic Databases. Table 3.1 exhibits the electronic databases we used in this work. We chose them because they are virtual libraries with an extensive collection of complete works and metadata from published research at conferences and journals of great importance to the academic community.

Table 3.1: Electronic digital libraries.

Databases	**Addresses**
ACM Digital Library	`http://dl.acm.org/`
Compendex (Engineering Village)	`https://www.engineeringvillage.com`
IEEE Xplore	`http://ieeexplore.ieee.org/`
Scopus	`http://scopus.com/`
Web of Science	`http://webofknowledge.com/`

Search String. To identify relevant papers regarding models on software evolution, we formulated a search string with terms related to the topic stated in this SLM. Initially, we defined the keywords "software evolution," "structure," and "model" as the main terms of our expression. Next, we searched for synonyms of these terms to refine this expression and identify relevant and coherent studies to answer the proposed research questions. We also used the symbol "*" in the terms to identify variations of the words, e.g., structure and structures.

Before defining the final string, we conducted a pilot search to identify relevant studies and decrease the recovery of unrelated studies to our topic. We included the terms "software maintenance" and "maintenance" during the pilot search. However, we identified that the inclusion of these words into our string returned many studies unrelated to the topic. Due to this reason, we decided not to include them in our search string. Besides, as our focus is models on software evolution in the context of the "Software Engineering" and "Computer Science" areas, we included these terms into our search string to reduce the number of documents not related to these areas.

Therefore, the final search string is defined as follows.

("software evolution" OR "system* evolution" OR "program* evolution" OR "evolution of software" OR "evolution of system*" OR "evolution of program*") AND ("architecture*" OR "design*' OR "structur*" OR "code*") AND ("model*") AND ("computer science" OR "software engineering")

Inclusion and Exclusion Criteria. The inclusion and exclusion criteria are applied to classify each primary study as a candidate to be included or excluded from the SLM [Kitchenham and Charters, 2007]. As an SLM may involve many studies, we limited the scope to select only complete papers and ignore short articles or documents classified as theses or dissertations. We decided to do so because the authors usually publish the studies regarding dissertations or theses as full papers. Besides, a short article usually presents emerging results. We must highlight that we consider full papers as documents containing six pages or more. Table 3.2 presents the inclusion and exclusion criteria we have defined in this study.

Table 3.2: Inclusion and Exclusion Criteria.

	Inclusion Criteria
1	Papers published in English
2	Full papers
3	Articles published in Computer Science and Software Engineering
4	Articles available in electronic format
5	Papers published in conferences and journals
6	Papers related to the topic investigated in this study
	Exclusion Criteria
1	Duplicate studies
2	Documents classified as tutorials, posters, panels, talks, lectures, round tables, theses, dissertations, book chapters, and technical reports
3	Papers that cannot be found

3.1.2 Execution

The execution phase consists of applying the search string to the electronic databases to identify candidate studies to be analyzed and filtering the analyses using the inclusion and exclusion criteria to select only the relevant studies. We carried out the search process of this SLM from June 20^{th}, 2019 to June 25^{th}, 2019 and updated it on November 23^{rd}, 2022. This update was conducted with the same search string and with the same datasets; therefore, it returned the same papers obtained in the first search process in 2019, plus the papers published after June 25^{th}, 2019 up to November 2022. In this chapter, we decided to describe the recent search process undertaken due to the exposed reasons.

Search Process. We did not define any constraint during the search and considered all studies returned by non-filter databases per publication year. Table 3.3 shows the results regarding the number of papers found in each electronic database. We obtained a total of 5,729 documents at the end of this process.

Table 3.3: Studies obtained after the search process.

Database	**Studies Returned**
ACM Digital Library	3,570
Compendex (Engineering Village)	738
IEEE Xplore	460
Scopus	545
Web of Science	416
Total	**5,729**

Papers Selection Process. As we identified many papers in the search phase, the selection process consisted of five steps. The steps focused on the inclusion and exclusion

criteria, including the concern with each study according to its content. We describe these steps as follows.

Step 1 - Exclusion of duplicate studies. In this step, we applied the first item of the exclusion criteria and removed identical studies, ensuring that only one register of a given paper remains. To do so, we analyzed the titles and authors of the documents and discarded the duplicate entries. This step removed 78 articles, resulting in 5,651 for Step 2 analysis.

Step 2 - Exclusion of documents that are not papers. It consisted of removing documents not classified as complete papers by applying the second item of the exclusion criteria and the second item of the inclusion criteria. Therefore, this step discarded the ones classified as tutorials, posters, panels, lectures, round tables, theses, dissertations, book chapters, technical reports, and short papers. We must highlight that we consider a complete article as one with six pages or more, and articles under six pages were considered short papers and removed here. This step released 2,434 documents, resulting in 3,217 studies for Step 3 analysis.

Step 3 - Metadata Reading. In this step, we analyzed the title and the abstract of the 3,217 papers obtained in Step 2 to select the ones relevant to this SLM. Besides, we also applied items 1 and 3–6 of the inclusion criteria and item 3 of the exclusion criteria. Although the paper's title and abstract provide an idea about the subject treated in it, it is sometimes necessary to read the article more in-depth to identify whether it is relevant. To avoid hasty decisions and exclude relevant documents, we classified as "dubious" the studies we could not select by reading the metadata. Step 4 analyzed such papers. At the end of Step 3, we identified 38 relevant articles already chosen for this SLM and 19 "dubious" ones.

Step 4 - Diagonal reading of dubious papers. This step aimed to review the "dubious" documents found in Step 3 to be included or not in this SLM. To identify these occurrences, we performed a diagonal reading of the papers. The diagonal reading analyzes the papers' introduction, topics, and conclusion to find more detail. At the end of this step, we concluded that 5 of the 19 "dubious" studies were relevant and included them in this SLM.

Step 5 - Snowballing. Searching in the electronic database does not guarantee that all relevant studies related to a particular topic will be retrieved. To mitigate this limitation, we carried out a snowballing procedure. *Snowballing* is a search approach that uses paper citations as a reference list to identify additional studies not found in the search process Wohlin [2014]. We may perform the snowballing in two ways: backward and forward. *Backward snowballing* refers to using the reference list of the papers to identify other studies. *Forward snowballing* refers to identifying new articles by analyzing the studies that cite a given study. We adopted the backward snowballing strategy and analyzed the reference list of 38 papers selected after Steps 3 and 5 chosen articles after

Step 4. We reviewed 1,786 references and appointed 28 new studies in this step.

In summary, Steps 1 - 4 received as input the following quantity of primary studies, respectively: (i) 5,729; (ii) 5,651; (iii) 3,217; (iv) 19 primary studies. Before reaching Step 5, we identified 43 relevant studies, 38 after Step 3 and 5 after Step 4. Finally, Step 5 received 1,786 references from the 43 papers already chosen in the previous steps. This step recovered 28 studies, resulting in 71 papers selected for this SLM. Figure 3.1 summarizes the selection process. The selected primary studies were analyzed and summarized to answer the research questions.

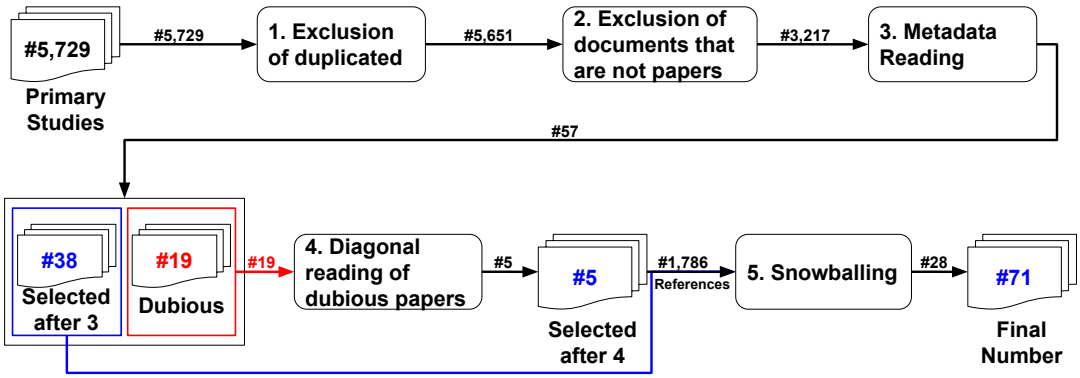

Figure 3.1: The filtering process carried out for selecting the primary studies.

It is important to highlight that the advisors of this Ph.D. dissertation participated in this process. More specifically, steps 1 and 2 were carried out by the author of this Ph.D. dissertation. Step 3 had the participation of the advisors. The author of this work was responsible for reading the papers' metadata. In this step, the advisors were responsible for analyzing the metadata of the papers that the author could not define as selected. In Step 3, the advisors were also responsible for analyzing whether the papers indicated to be selected by this work's author were eligible to be chosen. Step 4 also involved the participation of the advisors during the analysis and discussion to decide if the "dubious" papers could be selected after the diagonal reading. In Step 5, the author was responsible for analyzing the list of references of all selected papers until Step 4 to identify possible novel documents to compose the primary studies of this SLM. In contrast, the advisors were responsible for analyzing the papers the author pointed out and confirming whether they could be selected.

3.1.3 Data Extraction

We found 71 papers about software structure and source code evolution published from 1979 to 2022. We read and summarized them to extract preliminary information that answers the research questions. The author conducted the whole process supervised by the advisors of this Ph.D. dissertation. We made the data extracted from the primary studies selected for the SLM available on the website of our research group[1].

3.2 Results

This section presents the results of this SLM. We report the results in four subsections, each referring to the proposed research questions.

3.2.1 Categories of Proposed Models

This section answers **RQ1** – *What kinds of models on software evolution are proposed in the literature?*

We analyzed the papers selected in this SLM to group the ones with similar characteristics and determine categories of models proposed in the literature. We identified three types of software evolution models: (i) Characterization; (ii) Description; and (iii) Prediction. Figure 3.2 presents the number of papers obtained for each type of model.

Characterization: it is a category of models that aims to build a general abstraction that may represent the internal structure of a software system and provide an overview of how it evolves. There are 16 studies of this type. The design of this type of model is composed of internal components, which may provide a sufficiently precise representation of particular properties and their relationship. For instance, some software properties frequently defined and modeled by characterization models are classes, methods, and packages. Due to this, many studies have used graph-based approaches to build this type of model.

[1] `http://llp.dcc.ufmg.br/Publications/indexPublication.html`

Descriptive: refers to a category that aims to describe, understand, or explain how internal phenomena or events regarding the system behave over time. There are 18 studies of this type. Besides, it allows developers and researchers to characterize the systems' evolution patterns and extract properties or insights that may help them improve the system's structure and quality. The studies have usually applied statistical or mathematical methods to analyze the data and generate the model.

Prediction: the models of this category aim to predict the occurrence or presence of a particular event in the system's future. These models consider the evolution pattern of one or more predictor variables in determining the probability of some phenomena occurring, such as defects, increasing system size, and co-change. This category is the one that presented the higher number of studies, 37 in total.

Summary of RQ1. We identified three types of models for software evolution. They are (i) Characterization, (ii) Descriptive, (iii) and Prediction.

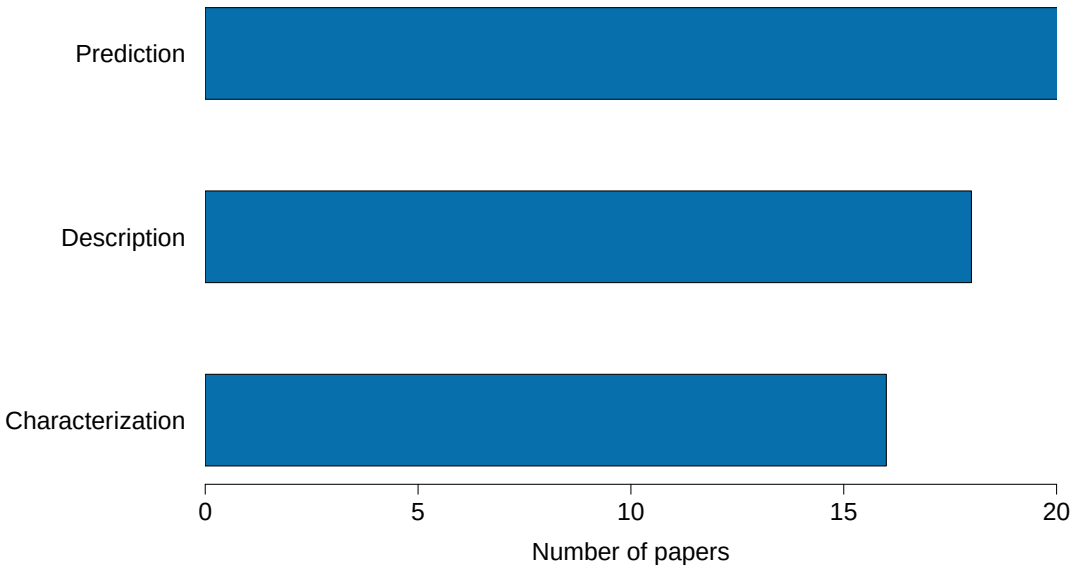

Figure 3.2: Number of papers by type of model.

3.2.2 Aspects of the software evolution models

This section answers ***RQ2*** – *Which software aspects have been considered in software evolution models?*

Table 3.4 summarizes the main objects of study of the proposed models. Regarding prediction, models mainly concentrate on the prediction of defects. There are also prediction models for changes, architecture evolution, clone evolution, effort, size, and coupling. However, they have not been explored as much as defect prediction. The Characterization models have been designed, in a vast majority, to represent the evolution of the internal structure of the software systems. Some studies have investigated how their internal components or types grow over time and how their connection changes and increases the system.

Table 3.4: The object of analysis of the software evolution models

Object of study	**Type of model**	**# of ref.**	**Ref.**
Defects	Prediction	11	[Graves et al., 2000, Arisholm and Briand, 2006, Aversano et al., 2007, Ratzinger et al., 2007a, Shatnawi and Li, 2008, English et al., 2009, Raja et al., 2009, Wu et al., 2010, Yang et al., 2014, Zong, 2020, Walunj et al., 2022]
Change		10	[Mockus and Weiss, 2000, Elish and Al-Rahman Al-Khiaty, 2013, Vaucher and Sahraoui, 2007, Amoui et al., 2009, Goulão et al., 2012, Yazdi et al., 2014, Shariat Yazdi et al., 2016, Bansal, 2017, Popoola et al., 2022, Silva et al., 2022]
Technical Debt		3	[Tsoukalas et al., 2019, 2020, Zozas et al., 2022]
Clone Evolution		2	[Antoniol et al., 2001, Pati et al., 2017]
Size		2	[Caprio et al., 2001, Herraiz et al., 2007b]
Trends in the evolution of Java systems		2	[Chaikalis and Chatzigeorgiou, 2015, Honsel et al., 2021]
Architecture evolution		1	[Trindade et al., 2017]
Characteristics of issues in Software repositories		1	[Izadi et al., 2022]
Code dependency		1	[Fan et al., 2021]
Effort		1	[Ramil and Lehman, 2000]
Refactoring		1	[Ratzinger et al., 2007b]
Reliability		1	[Amin et al., 2013]
Stability		1	[Bouktif et al., 2014]
Evolution of software structure	Characterization	13	[Myers, 2003, Jenkins S., 2007, Zheng et al., 2008, Pan et al., 2009, Wang et al., 2009, Pan et al., 2011, Bhattacharya et al., 2012, Ferreira et al., 2012, Wang et al., 2012, Petrić and Grbac, 2014, Li et al., 2016, Walunj et al., 2019, Liu et al., 2022]
Evolution of dependency networks		1	[Kikas et al., 2017]
Evolution of similar code fragments		1	[Huang et al., 2022]
Behavior of the evolution of the defects		1	[Vrankovic et al., 2019]
Explain the evolution of internal properties of the software	Description	15	[Woodside, 1979, Godfrey and Tu, 2000, Lehman et al., 2001, Ohlsson et al., 2001, Herraiz et al., 2005, Robles et al., 2005, Herraiz et al., 2007a, Koch, 2005, Leskovec et al., 2005, Koch, 2007, Kirbas et al., 2014, Ruohonen et al., 2015, Grbac and Mauša, 2016, Sousa et al., 2019a, 2021]
Analyze the evolution trends in software systems		1	[Capiluppi, 2003]
Describe the impact of design patterns over time		1	[Kermansaravi et al., 2021]
Describe the profile of the contribution of the user over the software evolution		1	[Honsel et al., 2016]

The Descriptive models proposed in the literature have aimed to explain the evo-

lution of intrinsic properties of software systems, such as size [Woodside, 1979, Godfrey and Tu, 2000, Lehman et al., 2001, Koch, 2005, Robles et al., 2005, Koch, 2007, Sousa et al., 2021], complexity [Herraiz et al., 2007a], coupling [Sousa et al., 2019a], defects [Kirbas et al., 2014], inheritance hierarchy [Sousa et al., 2021], maintainability [Ohlsson et al., 2001], reliability [Grbac and Mauša, 2016]. Moreover, some other studies have designed descriptive models to analyze the evolution trends in software systems, describe the impact of design patterns over time, and describe the users' contribution profile over the software evolution.

Summary of RQ2. Prediction models had considered mostly the evolution of defects. Characterization models have focused on providing an overview of the evolution of the internal structure of the systems. Finally, Descriptive models have been mainly designed to explain how the software systems' intrinsic properties behave over time.

3.2.3 Software Evolution Datasets

This section answers ***RQ3*** *– What are the main features of the datasets used in studies on software evolution?*

RQ3 aims to identify and characterize the datasets researchers have used to study software evolution. Answering this question is crucial as it may point out datasets that researchers may use in future studies and depict how diverse the samples are. We describe the datasets in three characteristics:
(i) the number of systems that compose them;
(ii) the type of dataset, i.e., if the dataset was proposed by the own authors of the paper or the paper's authors used a dataset made available by third parties in the literature; and (iii) the time frame considered by the dataset.

Initially, we investigated the distribution of the number of systems that compose the datasets. We summarize the results by category in Figure 3.3 and Table 3.5. By Figure 3.3 and Table 3.5, we conclude that the datasets are not composed of a large number of systems since half use less than three systems and 75% less than 10.

Although our results show that, in general, the datasets on software evolution are not composed of many systems, we also identified some outliers that indicate the existence of some datasets with a large sample of systems. For instance, we found datasets with 403,097 [Kikas et al., 2017], 8,621 [Koch, 2007], and 8,621 [Koch, 2005] systems,

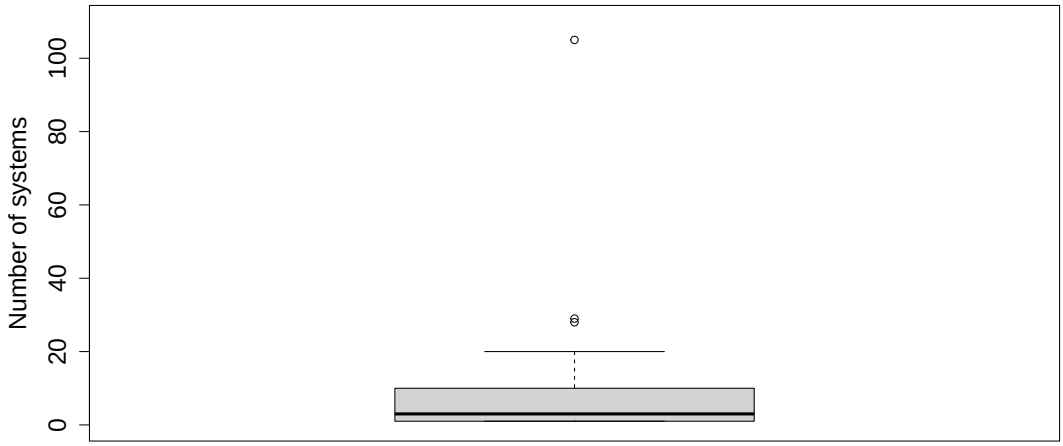

Figure 3.3: Distribution of the number of systems considered by the software evolution datasets.

Table 3.5: Distribution of the number of systems existing in the software evolution datasets.

0%	25%	50%	75%	100%
1.00	1.00	3.00	10.00	403,097.00

respectively. These values do not appear in the boxplot chart in Figure 3.3 because we limited its y-axis to 110 for better visualization. However, when we present the statistics of the complete set in Table 3.5, we may observe these broadly significant values. Among these three datasets, two of them [Koch, 2005, 2007] are unavailable online. On the other hand, the one proposed by Kikas et al. [Kikas et al., 2017] contains data about package dependency networks regarding the evolution of systems developed in JavaScript, Ruby, and Rust. It is publicly available[2] for access and use by other studies.

We classified the dataset used by each study as "own" or "third-party", i.e., a dataset created by the respective paper's authors and a dataset the author did not create. We identified that most of the studies used a dataset created by their authors, 62, which corresponds to 87%, and only nine used a third-party dataset.

The "third-party" datasets found in this SLM are (i) SourceForge [Koch, 2005, 2007]; (ii) COMETS [Sousa et al., 2019a, 2021]; (iii) GitHub [Kikas et al., 2017, Zozas et al., 2022]; (iv) Helix [Yazdi et al., 2014, Shariat Yazdi et al., 2016]; and (v) System 40 data [Amin et al., 2013]. COMETS[3], Helix[4], and System 40 data are software evolution repositories built to support studies on software evolution, while GitHub[5] and Source-

[2] `https://github.com/riivo/package-dependency-networks`
[3] http://java.labsoft.dcc.ufmg.br/comets/
[4] `http://www.ict.swin.edu.au/research/projects/helix/`
[5] `https://github.com/`

Forge[6] are source code hosting platforms with a version control system. Although they do not provide structured information about software evolution, it is possible to retrieve such details by creating scripts or tools.

Finally, we analyzed the data time frame the studies defined in their analysis. Initially, we collected the time frame regarding the information extracted from each system that composes the datasets. Then, for each specific date organized by year and month (YYYY-MM), we counted the monthly number of systems whose information refers to the respective date. For instance, suppose a paper analyzed information about a system "A" from 2010-01 to 2012-12. In this case, we increment one in the number of papers for 2010-01, 2010-02, 2010-03, until 2012-12 since the time frame has covered all these specific times. Figure 3.4 summarizes our analysis by a heat map chart where the lines represent the months, and the columns, the years.

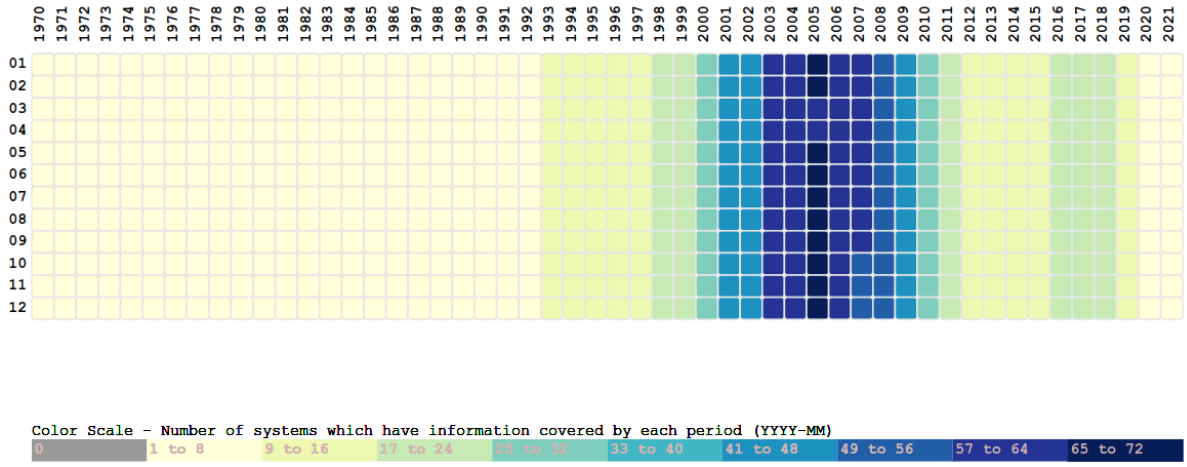

Figure 3.4: The number of systems considered in the studies by month/year.

Analyzing Figure 3.4, we realized that some datasets have considered updated data of software systems. For instance, the period between 2020 and 2021 comprises data from some datasets. However, the concentration in this period is shallow from 2000 to 2010, concentrating most of the data collected from the systems. Evolution data was collected between 2016 and 2018, but this period still covers a few systems. Therefore, this finding reveals the need to build datasets with more current data on software evolution.

[6] `https://sourceforge.net/`

Summary of RQ3. We identified that the software evolution datasets are composed of a small number of systems in general. We also found that the researchers usually define their software evolution datasets to carry out their studies on software evolution instead of using third-party datasets. Using one's own datasets to carry out empirical analysis is not a problem. However, the authors have not made available the dataset they built, which impairs the open science literature since other researchers can not reuse these data. Finally, we identified that most datasets are composed of data collected from 2000 to 2010.

3.2.4 Techniques to Build Software Evolution Models

This section answers ***RQ4*** – *What approaches have been applied for building models on software evolution?*

Several approaches have been used to build prediction models, which we summarize in Table 3.6. We detected six techniques: (i) ARIMA; (ii) Machine learning techniques; (iii) Regression techniques; (iv) Bayesian classifiers; (v) Dependency networks; and (vi) Markov Chain. Although they have been applied for the same purpose, they differ. For instance, *Regression techniques* and *ARIMA* are the most used strategies to design prediction models because of their power to estimate the relationship between independent and dependent variables. However, ARIMA uses the past value of a given variable to predict its future values.

In contrast, regression techniques already allow using one or more independent variables to predict future values for a different dependent variable. Machine learning algorithms, especially tree-based and classifier algorithms, such as J48, C4.5, and M5, are other techniques that have also been applied to design prediction models. According to Ratzinger et al. [2007a] and Krishnan et al. [2011], these algorithms efficiently define excellent and relevant predictors to the proposed models.

Bayesian networks, dependency networks, and Markov chains have also been used as strategies for predicting the occurrence of future events. A *Bayesian network* is a strategy that represents conditional independence between a set of variables via a directed acyclic graph model [Pearl, 2014]. It allows us to identify causality between variables via a cause-effect analysis and then build prediction models. *Markov chain* is a memoryless, homogeneous, stochastic process with a finite number of states that allows change from one state to another considering only the current state and not the sequence of events that

Table 3.6: Techniques used for designing the models on software evolution

Technique	Type of model	# of ref.	Ref.
Machine learning techniques	Prediction	14	[Aversano et al., 2007, Ratzinger et al., 2007b, Vaucher and Sahraoui, 2007, Amoui et al., 2009, Yang et al., 2014, Chaikalis and Chatzigeorgiou, 2015, Bansal, 2017, Tsoukalas et al., 2020, Silva et al., 2022, Zong, 2020, Fan et al., 2021, Izadi et al., 2022, Popoola et al., 2022, Walunj et al., 2022]
ARIMA		12	[Antoniol et al., 2001, Caprio et al., 2001, Herraiz et al., 2007b, Raja et al., 2009, Wu et al., 2010, Goulão et al., 2012, Amin et al., 2013, Yazdi et al., 2014, Shariat Yazdi et al., 2016, Pati et al., 2017, Tsoukalas et al., 2019, Zozas et al., 2022]
Regression Techniques		8	[Graves et al., 2000, Mockus and Weiss, 2000, Ramil and Lehman, 2000, Arisholm and Briand, 2006, Ratzinger et al., 2007a, Shatnawi and Li, 2008, English et al., 2009, Elish and Al-Rahman Al-Khiaty, 2013]
Bayesian networks		1	[Bouktif et al., 2014]
Complex networks		1	[Honsel et al., 2021]
Markov Chain		1	[Trindade et al., 2017]
Complex Networks	Characterization	12	[Myers, 2003, Jenkins S., 2007, Zheng et al., 2008, Pan et al., 2009, Wang et al., 2009, Pan et al., 2011, Wang et al., 2012, Petrić and Grbac, 2014, Li et al., 2016, Kikas et al., 2017, Vrankovic et al., 2019, Liu et al., 2022]
Graph Theory		3	[Bhattacharya et al., 2012, Walunj et al., 2019, Huang et al., 2022]
Little House		1	[Ferreira et al., 2012]
Regression Techniques	Descriptive	9	[Godfrey and Tu, 2000, Capiluppi, 2003, Koch, 2005, Robles et al., 2005, Koch, 2007, Kirbas et al., 2014, Ruohonen et al., 2015, Sousa et al., 2019a, 2021]
Statistical techniques		5	[Ohlsson et al., 2001, Herraiz et al., 2005, Leskovec et al., 2005, Herraiz et al., 2007a, Grbac and Mauša, 2016]
Markov models		2	[Honsel et al., 2016, Kermansaravi et al., 2021]
Mathematical methods		2	[Woodside, 1979, Lehman et al., 2001]

previously happened [Häggström et al., 2002]. Therefore, besides modeling the prediction of events as states, it uses distribution probability to model changes from one state to

another. Complex networks use the graph concepts to model the software structure as a network with nodes representing the internal software components. The relationships between the components are the edges [Newman, 2003]. The properties extracted from the software networks have been used to predict trends in software evolution.

Graph-based strategies have been predominantly used by studies that propose Characterization models. More specifically, they have been based on complex network concepts. As mentioned above, *graph-based techniques* allow modeling the internal structure of the software system as a network where the nodes refer to the internal components, such as classes, methods, or packages, and the edges refer to the relationship between the nodes, such as method calls. In this way, this modeling helps the users visually observe how the software structure has evolved. Besides, the complex networks provide metrics and properties that are an additional device in assessing the software evolution networks. Therefore, such facts explain why the researchers have chosen complex networks and graph-based approaches instead of other strategies to build characterization models.

Regarding description models, we identified that the main techniques used to build them are: (i) Regression techniques, (ii) Statistical techniques, (iii) Markov models, (iv) and Mathematical methods. Regression techniques are the most used strategy to build this type of model. Statistical techniques, such as correlation, descriptive analysis, and probability distribution, have also been beneficial for detailing the internal evolution of the systems. Although mathematical methods and Markov models are used less often, they are essential techniques to help researchers build description models.

Summary of RQ4. Table 3.6 summarizes the identified techniques we found to build software evolution models for each model category. As the main strategies, we highlight applying Machine learning, ARIMA, and regression techniques for building Prediction models. Regression techniques are also relevant for producing Descriptive models. Graph-based techniques have been used in the Characterization model emphasizing complex networks.

3.3 Discussion

The results of this SLM show that researchers have mostly explored prediction models. Besides, defects and changes are the predominant aspects investigated in prediction works. However, we identified other elements studied in the prediction context, such as technical debt and clone evolution. We observed that, in some cases, they do not

present a reasonable hit rate. For instance, Ramil and Lehman [2000] proposed a prediction model to predict effort in software systems. As a result, they obtained a model with errors of the order of 20 percent of the actual values. The appropriateness of the proposed models is restricted to homogeneous evolution segments as periods with relatively small variations in the level of effort applied. Then, situations like the one described in this example reveal gaps in the prediction models, which require other studies to assess their results. Besides, further studies are necessary to explore critical internal properties not identified by this SLM, such as coupling, cohesion, inheritance hierarchy, and others.

When analyzing the datasets used to build the software evolution models, in general, the datasets composed of at most ten software systems are used by at least 75% of the studies. This finding shows that description models are usually built for specific contexts via case studies. Then, the lack of models that describe the behavior of a particular aspect of software is a gap in the literature.

Another relevant characteristic this SLM reveals is the need for extensive and updated software evolution datasets publicly available in the literature. We identified some software evolution datasets reported by the works as COMETS, Helix and others, that are not too large or outdated. Because of this, most researchers decide to produce their dataset. However, even in those cases, the datasets were not made available. Therefore, all these facts evidence that the absence of a large and public dataset for software evolution studies is another gap in the literature that needs to be covered.

Finally, we analyzed the techniques used by the works to build the software evolution models. We identified many different methods, of which stand out machine learning, ARIMA, and regression techniques for prediction models; complex networks for characterization models; and Regression and statistical techniques for description models. The analysis of the methods carried out in this SLM can serve as a catalog and guide other researchers to define the procedures to build their models.

3.4 Threats to Validity

This section reports the threats to the validity of this study and the decisions we took to mitigate them.

Definition of the Search String. The search string definition is one of the main threats in an SLM since it defines the studies that will be analyzed, and relevant studies may only be returned if a search string is well defined. To mitigate this threat and avoid missing relevant studies, we searched several synonyms regarding our research's main terms and made several variations of the search string. We compared them and

conducted a pilot search before defining the final search string. The pilot search allowed us to calibrate the search string and choose one that returned relevant papers. We mitigated this threat to validity, and the defined search string returned as many relevant papers as possible.

Choice of the Electronic Libraries. The choice of electronic libraries is another factor that may impact the results of an SLM. Not all Software Engineering conferences are indexed in electronic libraries, so we may have missed relevant studies published in such conferences. We chose five electronic libraries important to the academic community to mitigate this threat. The choice of these libraries significantly reduces the chance of missing relevant studies. Besides, we ran a snowballing process during the selection phase of this SLM, i.e., we revisited the reference lists of the selected papers to recover studies that the SLM did not retrieve.

Selection Process. The selection process is a phase in an SLM where the studies retrieved by the search string are analyzed to remove the ones unrelated to the research topic. How we carry out this process may threaten the validity of an SLM because we may discard relevant studies. To mitigate this threat, we defined a sequence of five clear and strict steps, where the decision to toss a paper involved an analysis of the three authors from this study. Therefore, how we defined the steps in this process and the investigation was essential to alleviate this threat.

Generalization of the Results. The generalization of results is a troublesome threat to mitigate. We may not claim that we may generalize our results because we considered only papers written in English, and other relevant studies may be written in other languages. However, the primary vehicles of scientific publication in Software Engineering require articles written in English. For this reason, the choice of English was a decision that mitigates this threat.

Data Extraction. One of the paper's authors analyzed the selected articles and extracted the information supporting the answers to the research questions. This situation may be a threat to the validity of this study. To mitigate this threat, three authors discussed the data gathered from the papers and the decision to discard a paper.

3.5 Related Work

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

Breivold et al. [2010] conducted a systematic literature review on software evolution to extract an overview of the area. The authors concluded that the software evolution had

approached in four different ways: (i) software trends and patterns, (ii) evolution process support, (iii) evolvability characteristics addressed in OSS evolution, and (iv) examining open source software at the software architecture level.

Syeed et al. [2013] tried to compile the knowledge on software evolution via a systematic literature review. They analyzed 101 articles between the years 2000 and 2012 published in relevant Software Engineering venues. The authors characterized the area until 2012 and indicated gaps in the topic, e.g., the low presence of prediction studies focused on software evolution.

Vogel-Heuser et al. [2015] studied the state-of-art evolution of automated production systems to extract an overview of the area. Through this research, they identified the main lines of investigation inside this topic and provided a list of challenges and issues opened inside this theme.

Rasool and Fazal [2017] investigated the literature via a systematic literature mapping, focusing on open-source software evolution prediction and open-source software evolution process support. The goal of Rasool and Fazal [2017] was to study and analyze metrics, models, methods, and tools for open-source software evolution prediction and evolution process support. They identified 20 different categories of metrics, being that the most used is source lines of code (SLOC). Besides, they found that ARIMA is the most used model for building prediction models.

Marques et al. [2019] carried out a systematic literature review focused on software product lines (SPL) to identify and synthesize the characteristics supporting SLP evolution. They analyzed a total of 60 studies. Through the analysis of these studies, the authors have pointed out that there is still a gap in the literature regarding how the SPL evolves since the primary studies have not provided a consensus about the evolution.

Although the related works have investigated software evolution, they differ from our study in some points. Marques et al. [2019] and Vogel-Heuser et al. [2015] focused on particular contexts of software, software product lines (SPL), and automated production systems. Our study differs from these previous work because we did not focus on analyzing software evolution from a particular context. We considered studies that worked with software systems from any context.

Breivold et al. [2010], Syeed et al. [2013], and Rasool and Fazal [2017] carried out a systematic literature review (SLR) on software evolution concentrated in open-source software systems. Our study's main difference from these previous studies is that we did not limit it to open software. Besides, we focused our systematic literature mapping on compiling the state-of-art on software evolution models, while the previous studies focused on analyzing software evolution as a whole.

3.6 Final Remarks

This chapter presented a systematic literature mapping (SLM) aiming to investigate the state-of-the-art on software evolution models. Mainly, we aimed to identify: (i) the main types of models that the researchers have built; (ii) the purpose for which the software evolution models have been built; (iii) the characteristics of the datasets that have been used to build the software evolution models; and (iv) the techniques that have been used to create the models.

This SLM comprised the analysis of 71 works regarding models on software evolution. We identified three categories of models, and they are (i) Prediction; (ii) Characterization; and (iii) Description. This SLM provided the compilation of the main insights on software evolution models. The results of this study lead us to the following main conclusions:

- There are three software evolution models proposed in the literature: Characterization, Description, and Prediction. The Prediction model is the prevalent type.
- Prediction models mainly concentrate on predicting defects and changes. Characterization models have considered the evolution of the internal software structure. Descriptive models have been designed to explain how the intrinsic properties of software systems evolve.
- We identified that the researchers have invested in proposing software evolution datasets instead of using existing ones in the literature. This behavior may be explained by the fact that only some software evolution datasets are publicly available in the literature. Besides, the datasets usually contain data from a few software systems and consider old versions of those systems.
- Many techniques and strategies have been used to build software evolution models. Machine learning, ARIMA, and Regression techniques are the most used in prediction models. Descriptive models also used Regression and statistical methods. Characterization models usually apply graph-based techniques.

Based on the findings from this SLM, most of which we summarized in sections 3.3 and 3.6, motivated us to build a comprehensive software evolution dataset containing temporal data of open-source Java systems, and the novel method for software evolution analysis based on time series, described in Chapter 4 and Chapter 5, respectively.

Chapter 4

A Time Series-Based Dataset of Open-Source Software Evolution

Many studies have investigated how the process of software evolution occurs [Meyer, 1996, Daly et al., 1996, Cartwright, 1998, Harrison et al., 2000, Godfrey and Tu, 2000, Capiluppi et al., 2004a,b, Capiluppi and Ramil, 2004, Robles et al., 2005, Herraiz et al., 2006, Izurieta and Bieman, 2006, Capiluppi et al., 2007, Herraiz et al., 2007a, Koch, 2007, Tempero et al., 2008, Nasseri et al., 2008, Gonzalez-Barahona et al., 2009, Eski and Buzluca, 2011, Abuasad and Alsmadi, 2012, Couto et al., 2012, Alenezi and Zarour, 2015, Hatton et al., 2017, Singh and Ahmed, 2017], but there is a lack of software evolution data to be analyzed. Some works have made efforts to provide datasets and make them public [D'Ambros et al., 2010, Vasa et al., 2010, Tempero et al., 2010, Couto et al., 2013, Gousios, 2013, Ma et al., 2019]. Nevertheless, they are not too large, are outdated, or do not provide time series about software metrics.

Therefore, we built a comprehensive software evolution dataset containing temporal data from open-source Java systems. We defined a methodology composed of four steps: (i) we determined a selection process with requirements for selecting a system; (ii) we built an approach to make releases from the software systems; (iii) we extracted static software metrics for the systems' releases; and (iv) we generated the time series of the metrics values. We used GitHub[1] to collect the data of the software systems. We have applied this dataset to a study to model the software evolution process. In this chapter, we present this dataset. We made the dataset available so that other studies on software evolution may use it[2].

It is important to highlight that the main goals with the creation of this dataset are: (i) cover the gap existing in literature regarding lack of large and updated software evolution datasets; (ii) support the studies carried out in this work even it considering only a small part of data present into the dataset; and (iii) making the path of other researchers in the search for data to analyze software evolution less arduous, given the difficulty of publicly available data for empirical analysis. Due to these reasons, we built

[1] `https://github.com/`

[2] `https://brunolsousa.github.io/software-evolution-dataset/index.html`

a dataset containing temporal data regarding 46 software metrics from 46 Java software systems.

We organized this chapter as follows. Section 4.1 describes the process we follow to build the dataset. Section 4.2 presents some related works. Section 4.3 discusses the threats to validity and the decisions we took to mitigate them, and Section 4.4 concludes this chapter.

4.1 The Construction of the Dataset

This section describes the steps we followed to build the dataset. Step 1 reports how we selected the subject projects. Step 2 explains how we made the systems releases. Step 3 details how we extracted the software metrics from each release generated for the software systems, and Step 4 describes the time series generation process. Figure 4.1 presents an overview of these four steps.

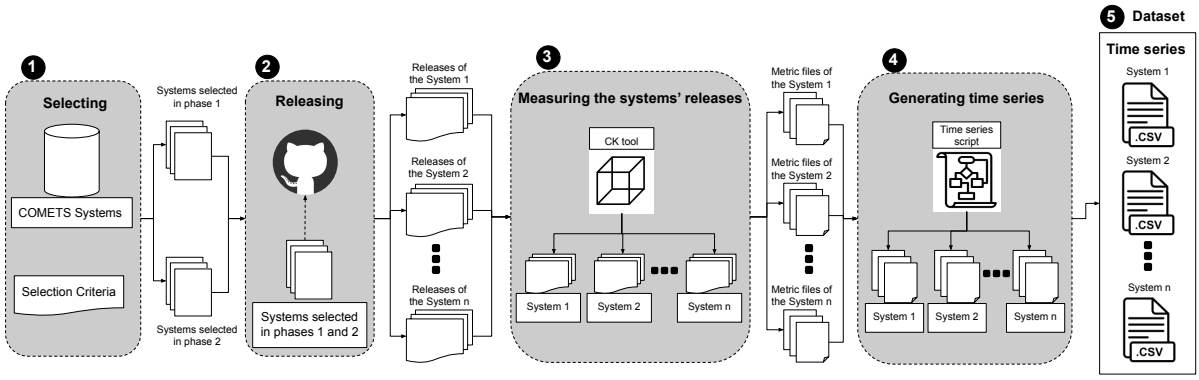

Figure 4.1: The dataset creation process.

Step 1 - Selection of the Subject Systems. We identified systems using a selection criterion composed of two phases. First, we selected nine systems from COMETS [Couto et al., 2013]. We included them because COMETS is an important time series dataset and contains relevant systems. However, it was created in 2010 and is outdated. Second, we based on the information from GitHub to define the characteristics a system should have to be selected. Therefore, to choose a software system, we considered the following aspects:

1. **Programming Language:** The system needs to be developed in Java. We concentrated on Java because it is a programming language widely used in software engineering empirical studies and is highly popular among developers.

2. **Popularity:** This requirement ensures the selection of popular software systems. We considered here the number of stars from the projects in 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 5,000 commits. We defined 5,000 as the minimum limit of commits for a system because this was the lower value identified in the software systems selected in the first phase.

4. **Lifetime:** To include a system in the dataset, it must have at least five years.

5. **Not be deprecated:** The year of the last commit in the project should be 2020. This requirement avoids selecting systems that the developer community has abandoned.

We implemented a Python script using a REST API from GitHub, applying the criteria defined to select the systems. We obtained 46 software systems; nine belong to COMETS, and 37 were selected in the second phase. It is essential to highlight that *TV-Browser* considered in COMETS was discontinued in 2013. Therefore, as it is not updated, we did not consider it.

Step 2 - Extracting the Software Systems' Releases. We carried out a process to extract the systems' source code in a version way. We based on GitHub's information about the software systems' whole lifetime and defined an interval to delineate a release. Then, we followed the same methodology used by Couto et al. [2013] during the building of COMETS and formalized a release as source code information of software regarding bi-weeks, i.e., 14 days.

We implemented four scripts in Python performing this step. 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[3].

We defined the following approach to get the released source code of the systems. We identify the repository of a particular system on GitHub by providing the complete name of this repository for our approach. It is essential to highlight that the full name of the repository is always composed of a user's name followed by a bar ("/") and the name of the system repository. For instance, if we want to access the *Eclipse JDT CORE* repository, we need to inform the script of "eclipse/eclipse.jdt.core". 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 creates a folder using the project's name and clones the repository inside its respective folder. With the support of PyGitHub, our script analyzes the data and extracts its release list, considering the project's life timeframe on GitHub. At this

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

moment, the release list of the project is only a sequence of bi-week periods that indicate the beginning and end of each release. The script downloads the source code of each release, considering the last commit from the bi-week interval, inside the project's folder, viewing its release list, creating a sub-folder for each release, and storing the respective source code inside this sub-folder. Figure 4.2 summarizes our script's sequence of steps.

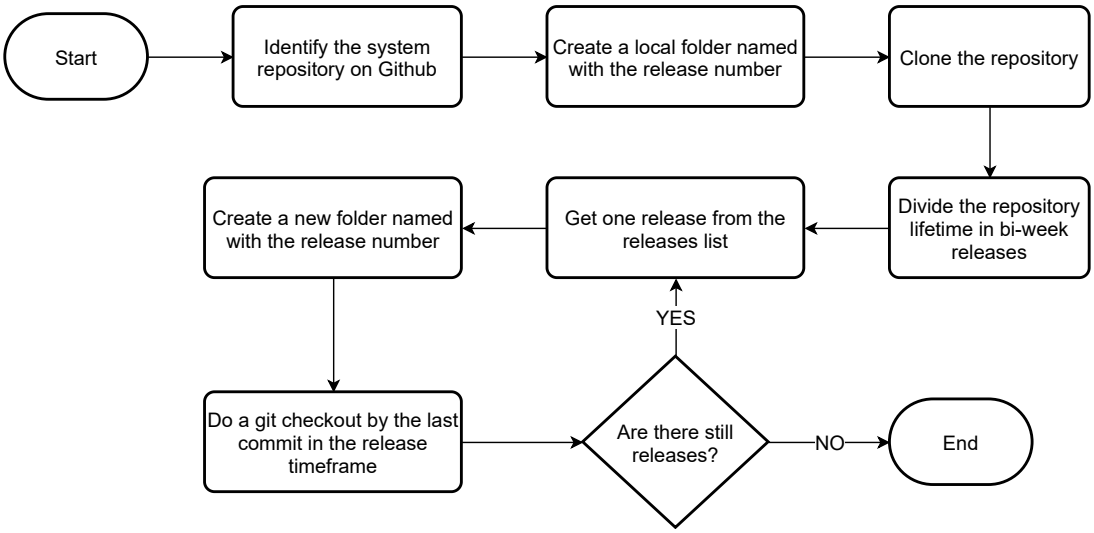

Figure 4.2: Process of the extraction of the systems' releases.

Step 3 - Collecting Metrics. We computed static metrics of the releases of the 46 selected systems. We collected 46 software metrics that characterize several aspects of the software, such as size, cohesion, inheritance hierarchy, coupling, complexity, and others. 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 [Aniche, 2015]. We decided to use it for its ease of installation, good documentation, and completeness regarding the number of software metrics. Although the 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 many information about the software systems. CK TOOL exports the metrics values in *CSV* files.

Step 4 - Generating Time Series. We considered the same pattern defined in [Couto et al., 2013, Couto, 2013] for generating the time series. This pattern consists of defining *CSV* files for each metric M and each system S, where the lines represent the classes of S, and the columns represent the versions. Each cell *(c,r)* consists of the metric value of class c in the release r. Then, each generated *CSV* contains a 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. 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. To perform this separation, we followed the same COMETS [Couto et al., 2013] method, which consists of

filtering the classes considering their directory when generating their time series to remove these classes. We did not generate time series from classes kept in a directory that started with "test", or from directories with "test" as part of its complete name. In this step, we created more Python scripts, besides the four created in Step 2, to automate the process of time series generation.

Table 4.1 presents the 46 open-source Java systems that compose our dataset. We publicly make the dataset available and provide additional information on our supplementary website [SEDataset, 2021]. Besides, we created five Python scripts for collecting the system's data from GitHub and building the dataset. All scripts used to construct the dataset are publicly available as supplementary material[4].

4.2 Related Work

The literature has produced datasets for supporting studies on software evolution. Some are publicly available [D'Ambros et al., 2010, Vasa et al., 2010, Tempero et al., 2010, Couto et al., 2013]. In contrast, others are not [Robles et al., 2005, Herraiz et al., 2006, Raja et al., 2009, Darcy et al., 2010, Grigorio et al., 2015]. This section provides an overview of these two kinds of datasets and discusses how our study differs from the related works.

Public datasets. Aiming to support empirical studies on software evolution and evaluate bug prediction models, D'Ambros et al. [2010] proposed a benchmark containing data on defects in software systems over their lifetime. The D'Ambros dataset comprises time series metrics values of five open-source Java systems. Vasa et al. [2010] also proposed a dataset that provides temporal information on source code static metrics' values. It comprises 40 open-source Java systems and contains time series information on many software metrics. However, this dataset did not include several relevant software metrics, such as CK metrics [Chidamber and Kemerer, 1994] and coupling. Tempero et al. [2010] built a corpus composed of 111 software systems for supporting empirical studies on software engineering, named Qualitas Corpus, containing evolution data of 14 systems and did not represent these data as time series. Couto et al. [2013] proposed a dataset named COMETS, composed of a time series of 17 well-known software metrics data of defects in 10 Java software systems. Although these datasets contain software metrics data, they were created years ago and are outdated. Gousios [2013] constructed a public dataset named GHTorrent, which contains information about many open-source systems. It contains the primary information about a software system from GitHub, such as repository

[4] `https://github.com/BrunoLSousa/DSTool`

Table 4.1: 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/MinecraftFo	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

name, commits, and pull requests, essential for extracting insights about the software systems. However, it does not provide time series from the software systems. Ma et al. [2019] proposed World of Code (WoC), a public dataset containing information about 1.6 billion commits made by authors of repositories from GitHub. This dataset has versioned

data about the software. However, it does not provide time series regarding the source code metrics from the systems.

Not publicly datasets. Robles et al. [2005] studied how open-source software systems evolve from the size perspective. For this purpose, they created a dataset containing time series from size metrics of 20 relevant software systems. Herraiz et al. [2006] investigated how software systems evolve in size from different perspectives. They built a dataset composed of time series regarding size software metrics. Their dataset contains temporal information from 13 open-source software systems. Raja et al. [2009] carried out their dataset with time series regarding defects in software. Their purpose was to analyze how defects evolve in open-source systems. The proposed dataset comprises eight open-source Java systems and contains only information about defects. They spaced out the time series considering 30 days as the release interval. Darcy et al. [2010] conducted an empirical study to identify the pattern that open-source software systems follow over time. They built a dataset composed of 108 software systems to perform this analysis. Their dataset contains temporal information on size and complexity metrics, which were not publicly available. Grigorio et al. [2015] studied the relationship between complexity metrics and the abandonment of open-source systems. They created a dataset of 10 open-source systems and provided time series of complexity metrics. Each period presented in their dataset refers to a release defined by the project owners in SourceForge[5].

This work proposes a software evolution dataset composed of 46 relevant open-source Java systems hosted in GitHub. Our dataset contains a time series of 46 static metrics for each system. Even though several studies proposed software evolution datasets, not all of them are available to be used in further empirical studies [Robles et al., 2005, Herraiz et al., 2006, Raja et al., 2009, Darcy et al., 2010, Grigorio et al., 2015]. Although there are some of them made public [D'Ambros et al., 2010, Vasa et al., 2010, Tempero et al., 2010, Couto et al., 2013, Gousios, 2013, Ma et al., 2019], they do not contain software metrics data and are outdated. Our present work differs from the studies described above because our dataset (i) is public, (ii) contains recent information on system evolution, (iii) provides software metrics time series, and (iv) to the best of our knowledge, it is the largest dataset of metrics time series, both in number of software metrics and systems.

4.3 Threats to Validity

This section presents the threats to the validity that we identified while building the dataset and discusses the major decisions we made to mitigate them.

[5] `https://sourceforge.net/`

We defined a selection criterion and considered data from 46 open-source Java systems hosted on GitHub to build our time series dataset. As GitHub is a significant control version platform with many software systems, our dataset may be considered limited to the number of systems. However, collecting time series from open-source Java systems requires considerable effort. Besides, no other public time series dataset as extensive as the one we propose is public available. Therefore, our dataset can be advantageous to researchers and practitioners in carrying software evolution studies.

To carry out the data collection, we used tools and scripts. We used CK Tool [Aniche, 2015] to extract the metrics' values from the systems' releases, and we also built Python scripts to release the source code of the systems and generate the time series. Although tools automate and facilitate the data collection process, they configure a threat to validity since tools may have errors. We used a well-evaluated and tested tool to mitigate this threat. We also manually checked the results exported by CK Tool and the Python scripts to ensure they produced results as expected.

4.4 Final Remarks

This chapter presented a comprehensive software evolution dataset with temporal information on 46 static software metrics of 46 open-source Java systems. We defined a methodology composed of four steps to build the dataset reported in this work. First, we determined the aspects we consider relevant to choose the systems. Second, we made the releases from the source code of the systems. Third, we collected the static metrics from the releases. Finally, we generated the time series in the fourth step.

Although some literature efforts aim to provide data to support software evolution studies [D'Ambros et al., 2010, Vasa et al., 2010, Tempero et al., 2010, Couto et al., 2013, Gousios, 2013, Ma et al., 2019], the datasets built so far do not encompass many systems or time series, and some of them are outdated. The dataset created in this work fills this gap because it contains many systems and software metrics, provides time series, and contains recent information about the systems.

We used the dataset described in this chapter to construct the models for software evolution analysis and prediction proposed in this dissertation, as we describe in chapters 5 and 6.

Chapter 5

A Method to Model and Predict Software Evolution

This chapter presents the method we defined to model and predict the evolution of internal software quality attributes. The proposed method analyzes time series corresponding to software metrics of the four internal software attributes considered in this work: coupling, cohesion, inheritance hierarchy, and class size. The method comprises two phases: behavior analysis, presented in Section 5.1, and trend analysis, described in Section 5.2. Our method uses some statistical tests. Therefore, it is essential to mention that in all the statistical analyses, results are considered significant at a 5% level.

5.1 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 5.1 shows the steps of this phase.

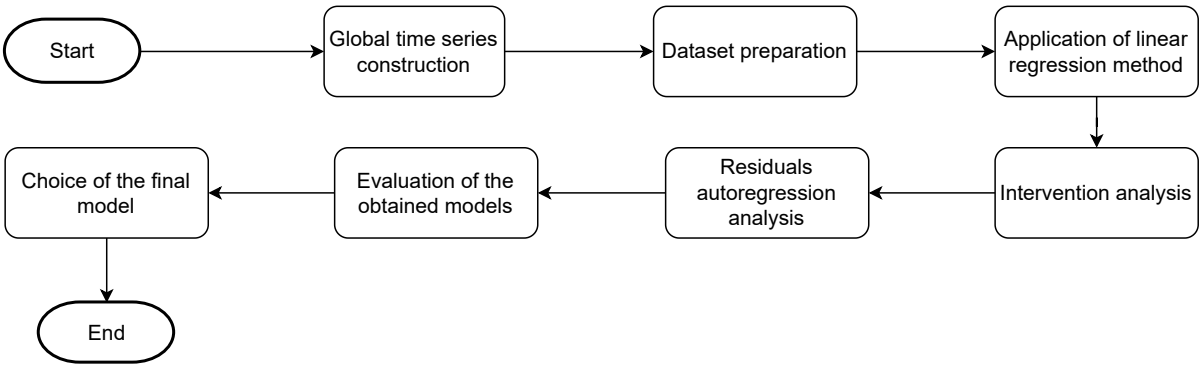

Figure 5.1: Steps of the behavior analysis phase.

We describe the behavior analysis steps as follows.

Step 1 - Global time series construction. We considered only metrics at the class level. As each class consists of a time series, a system is a component of many time series regarding a particular metric. Then, this step aims to reduce the several time series existing for a given system in a global measure that represents that characteristic globally. This step aims to create a global measure for each analyzed software system by taking the arithmetic average of their metric values. We 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. We used the training data 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. To divide our dataset into training and test subsets, we carried out pilot tests varying the percentages of the training subset between 60% and 90% and the test subset between 40% and 10%. During these tests, we observed that a training subset lower than 80% of the size of the time series would not be representative for extracting models from some time series. Then, we defined that our training subset comprises 80% of the periods belonging to the time series. The test subset comprises 20% of the periods 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 [Draper and Smith, 1981] 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 [Antoniol et al., 2001, Caprio et al., 2001, Raja et al., 2009, Yazdi et al., 2014]. The ARIMA technique requires that the time series periods be collected over a well-defined time scale, such as days, months, or years [Box and Jenkins, 1976]. In contrast, linear regression does not require the time series periods 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;

(vi) logarithmic at Degree 3.

We considered all these models to identify which best 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* [Wei, 2006], a technique that evaluates and measures the effects 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 affecting 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 [Bowerman and O'Connell, 1993]. Using linear regression to model time series may lead to autocorrelated error terms. *Autocorrelation* consists of a serial dependence between the values of adjacent periods of a time series [Cowpertwait and Metcalfe, 2009]. Suppose our models do not incorporate a component to deal with this serial dependence. In that case, the coefficient estimates and their standard errors may be wrong, and the model will not correctly represent the time series [Bowerman and O'Connell, 1993]. Step 5 evaluates the models' errors and removes autocorrelation by modeling them via autoregression. *Autoregression* is a process that uses periods from previous time steps to model the value at the next time [Cowpertwait and Metcalfe, 2009]. After modeling the residuals, we included 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 [Miles, 2014].

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. We composed our protocol of three criteria, Relevance, Coverage, and Simplicity, which evaluate a different aspect of the model.

1. **Relevance.** We selected the generated models with $\overline{R}^2$ higher than or equal to 80% since this percentage already defines models with good adjustment.

2. **Coverage.** We selected the type of models that generated relevant models for the most number of software systems inside our sample. For instance, suppose we are evaluating linear and quadratic models for a sample of 10 software systems. Also, suppose that the relevance criteria pointed out that four and six systems presented $\overline{R}^2$ higher than or equal to 80% for the linear and quadratic models, respectively. Thus, applying the coverage criteria, we would select the quadratic model as the better since it generated relevant models and had greater coverage for the analyzed systems compared with the linear.

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.

5.2 Trend Analysis

This section presents the second phase of our method, the trend analysis. This phase aims to analyze the percentage of classes directly affecting software systems' growth and decrease of a given internal attribute. It consists of seven steps, and we describe them as follows.

Step 1 - Data organization. When a given class is absent in a particular system period, 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 periods, 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 small sub-series. Figure 5.2 illustrates this phenomenon. In this example, the class *NonstrictReadWriteCache* was removed and reintroduced before the 100^{th} system's period. The concept of *ghost classes* is a legitimate software evolution pattern that arises in software development and maintenance. Although it is important to recognize the legitimacy of ghost classes as a software evolution pattern, it is also important to highlight

that considering *ghost class* may introduce bias in our analysis. When a class does not appear in a release and appears in the subsequent release, as it is happening close to the 100^{th} period in Figure 5.2, it generates a break in the time series. Trend tests work only with continuous time series, i.e., 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, although we recognize their legitimacy inside the software evolution process.

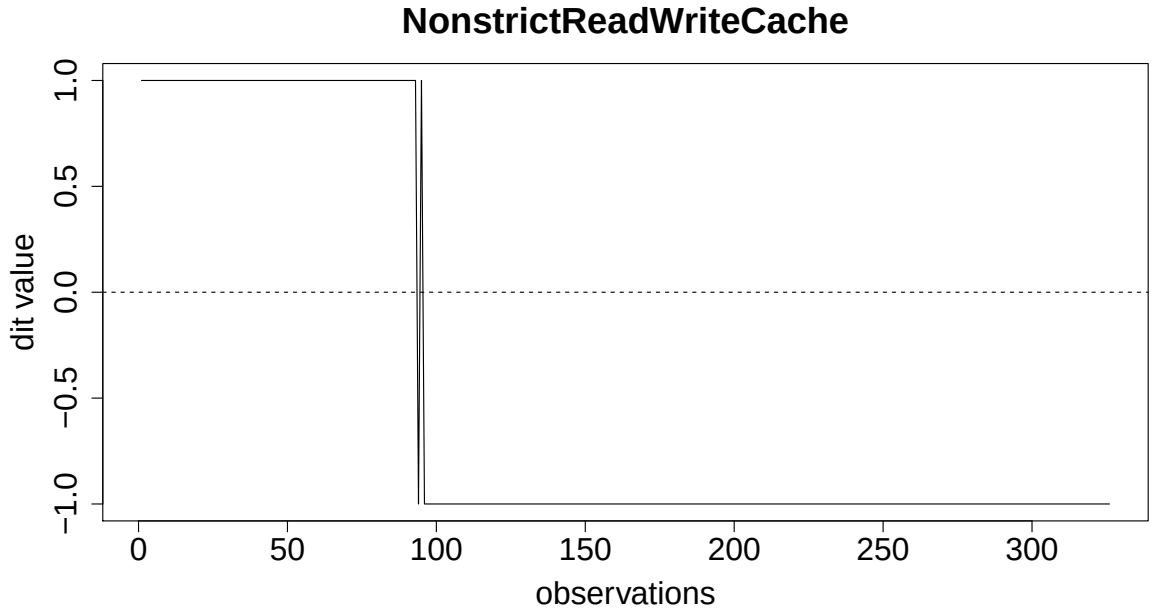

Figure 5.2: Time series of a ghost class.

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 [Kendall, 1975];

(ii) Cox-Stuart [Morettin and Toloi, 2006];

(iii) Wald-Wolfowitz [Morettin and Toloi, 2006].

We chose them because they have been pointed out as applicable and efficient [Kendall, 1975, Morettin and Toloi, 2006]. We considered the following hypotheses:

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

Even when choose efficient trend tests, the statistical tests may contain weaknesses and be prone to errors. 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 periods in some of them. Therefore, we decided to only analyze and apply the trend tests in time series with ten or more continuous periods. Times series with less than ten measures are disregarded and classified as having no trend.

Step 4 - Identification of autocorrelated time series. The Mann-Kendall test is sensitive to the presence of autocorrelation. When the original version of the Mann-Kendall test is conducted in an autocorrelated time series, it may generate false positives or false negatives [Hamed and Rao, 1998]. 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 [Box and Jenkins, 1976]. It describes how well a given value is related to its past periods. *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 [1998] 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.

- **Upward trend.** It is a pattern whose distance between the trend line and the x-axis increases over the x-axis
- **Downward trend.** It is a pattern whose distance between the trend line and the x-axis decreases over the x-axis

- **Undefined trends.** They are cases that do not follow a clear pattern or whose values of the first period are equal to the last.

5.3 Final Remarks

In this chapter, we described our two-phase method to analyze the metrics times series of the software systems. The first phase consists of extracting the global time series of the metrics obtained from the systems and modeling them by applying linear regression techniques. This phase aims to identify which model best describes the evolution of the systems' internal characteristics. Using only the standard linear regression to model auto-correlated data and time series with structural breaks does not provide good results and efficient models. Therefore, as a differential of our method, we defined some adjustments, such as intervention and autocorrelation analyses (steps 4 and 5), to treat these problems and ensure functional model production.

The second phase of our method consists of applying statistical trend tests to the time series of the classes from the software systems. This phase aims to identify the components with growth and decrease trends and, consequently, the ones that directly impact the increase and decrease of an internal attribute in a software system over its evolution process.

Chapter 6 reports the empirical study we conducted to analyze and predict software evolution by applying the method described in this chapter.

Chapter 6

Empirical Analysis of Software Evolution

This chapter reports the empirical analysis we performed to investigate how object-oriented software systems evolve from the coupling, cohesion, size, and inheritance hierarchy perspective. We analyzed the time series of six software metrics to represent these internal attributes. The time series are those provided by our dataset described in Chapter 4. We considered fan-in and fan-out to represent coupling, LCOM (Lack of Cohesion) and TCC (Tight Class Cohesion) to measure cohesion, NOA (Number of Attributes) and NOM (Number of Methods) to represent the size, and DIT (Depth of Inheritance Tree) and NOC (Number of Children), to measure inheritance hierarchy.

Fan-in is the number of classes that use a given class, while fan-out is the number of classes used by a given class [Lee et al., 2007, Sommerville, 2012]. As we described in Section 2.1, there are many static and dynamic object-oriented software metrics for coupling, such as Yacoub's metrics [Yacoub et al., 1999], Arisholm's metrics [Arisholm et al., 2004], Mitchell's metrics [Mitchell and Power, 2003, 2004, 2005, 2006], DCM (Dynamic Coupling Metric) [Hassoun et al., 2004a,b, 2005], and Gupta's metrics [Gupta and Rao, 2001]. We decided to use fan-in and fan-out because they consider the method invocations and references to 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 [Chidamber and Kemerer, 1994]. NOA and NOM are size metrics, and they refer to the number of attributes and methods of a class, respectively [Lorenz and Kidd, 1994]. 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 [Godfrey and Tu, 2000, Capiluppi et al., 2004a,b, Robles et al., 2005, Herraiz et al., 2006, Izurieta and Bieman, 2006, Capiluppi et al., 2007, Herraiz et al., 2007a, Koch, 2007, Gonzalez-Barahona et al., 2009, Darcy et al., 2010, Hatton et al., 2017]. Moreover, using NOA and NOM to analyze the size evolution in object-oriented software allows us to provide

an evolutionary view of this internal attribute from the perspective of data and services provided by the classes.

LCOM and TCC are cohesion metrics. While LCOM measures the lack of cohesion between methods of a class [Chidamber and Kemerer, 1994], TCC indicates the cohesion of a class via direct connections between visible methods [Bieman and Kang, 1995, Bär et al., 1999, Panas et al., 2005]. We chose these two metrics because there is no consensus for measuring cohesion yet. 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 properly analyze cohesion evolution [Ralph and Tempero, 2018].

We organize this chapter as follows. Section 6.1 presents the research questions we proposed to investigate throughout these empirical analyses. Section 6.2 details the evolution of the analyzed software's internal attributes. Section 6.2 examines how the coupling and size measurements evolve from the point of the relation between their metrics. Section 6.4 investigates the percentage of the studied software systems that contribute to increasing or decreasing the software's internal attributes over their evolution. Section 6.5 analyzes the accuracy of our time series approach concerning prediction. Section 6.6 summarizes the evolution properties we extracted, considering the observations identified during the answer to the research questions. Section 6.7 discusses some practical implications of the empirical analysis and software evolution properties identified in this chapter. Section 6.8 discusses the main threats to the validity of this empirical study and the decisions to mitigate them. Section 6.9 concludes this chapter.

6.1 Research Questions

This empirical analysis aims to study the evolution of the software's internal attributes 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 presented as follows.

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

This research question analyzes how the software's internal attributes evolve and identifies the patterns that better describe their behavior. We applied the behavior analysis phase of our method, described in Section 5.1, to the global time series of the software's internal attributes metrics to model and extract the best evolution pattern that represents

them. We carried out this analysis for each internal attribute studied in this work.

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

This research question investigated coupling and size. In the case of coupling, we analyzed the evolution of the relationship between fan-in and fan-out. This analysis will show if 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 will reveal if the number of methods and attributes of the classes evolves in a related pattern. Regarding inheritance hierarchy, we compared the values evolution of DIT and NOC from the classes of *Eclipse JDT Core* to detail preliminarily how the inheritance hierarchy has evolved inside 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, i.e., the internal cohesion of a class.

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

This research question aims to identify the percentage of classes in a software system responsible for increasing and decreasing the internal attributes over software evolution. To answer this research question, we applied the trend analysis, described in Section 5.2 as the second phase of our method, into the original time series of each class of the analyzed software systems to map the ones that have a growth trend and the ones that have 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. For this purpose, we divided our dataset into two parts: training data and test data. With the training data, we generated the models. We extracted the errors obtained for the models in each type with the test data: short-term and long-term forecasts. We compared the predictions for each model identified using the predicted mean square error (PMSE).

6.2 Evolution Properties at the System Level

This section answers **RQ1.** *Which model better describes the evolution pattern of the software systems' internal attributes?*

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

We applied the first phase of our method (Section 5.1) to the global time series metrics to evaluate how they evolve. As we perform a forecasting analysis in Section 6.5, the time series were 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. We did not include the time series charts in this document. However, we made them publicly available[1].

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 [Conover, 1998] to identify any difference between the proportion of growth 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:

- **H_0:** the proportion of growth and decrease is the same.
- **H_1:** the proportion of growth and decrease differs.

Table 6.1 shows the number of systems for each metric with a growth or decrease pattern. In Table 6.1, we also report the signal test results for each metric.

Table 6.1: 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	0.46139	0.02590	0.00164	0.00164	0.18392	0.05408	0.46139	1

In the results shown in Table 6.1, only 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 has increased in fan-in and fan-out, and the inheritance hierarchy has increased in breadth. Also, the results shown in Table 6.1 reveal that, for TCC, there are more systems with a growing pattern of this metric than systems with a decreasing pattern.

[1] `https://github.com/BrunoLSousa/SupplementaryMaterialSPEResearch`

Although this behavior is not significant at a 5% level, it would be considerable at a 10% level. TCC measures the internal cohesion of a class, considering only visible methods. Then, this result indicates that the dataset classes have become more cohesive over their evolution.

Regarding DIT, LCOM, NOA, and NOM, the signal test has not pointed out a significant 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 global inheritance hierarchy from the perspective of depth and size in terms of growth or decrease. However, a descriptive analysis from Table 6.1 shows that most systems have a decreasing pattern for DIT, NOA, and NOM.

After identifying the software metrics' evolution patterns, we modeled their global evolution using regression techniques. We applied our evaluation protocol, Section 5.1, considering the $\overline{R}^2$ values obtained from the models to detect which model better characterizes their global evolution. In Apendix A, tables A.1 – A.8 summarize the $\overline{R}^2$ scores computed for the models fitted regarding analyzed metrics internal attributes. 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 A.1 – A.8 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. It is important to highlight that all extracted models are available on the research website[2].

Analyzing the results obtained regarding DIT models in Table A.1, 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 A.2 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.

[2] `https://brunolsousa.github.io/site_tese/`

Tables A.3 and A.4 show 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*, *MinecraftForge*, *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.

Tables A.5 and A.6 show the results of modeling LCOM and TCC time series. 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, i.e., 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 describes the evolution of TCC.

Finally, Table A.7 and A.8 report 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 an appropriate 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 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.

6.3 Relation between the Metrics' Evolution

This section answers **RQ2.** *How does the relation between internal attribute metrics behave throughout the evolution of software systems?*

We analyze RQ2 for coupling, inheritance hierarchy, and size. To organize the discussion of RQ2, we categorize the answers as follows. Section 6.3.1 discusses the relation between fan-in and fan-out using the idea of necessary and unnecessary coupling. Section 6.3.2 discusses the connection between DIT and NOC via a case study to detail how the hierarchy inheritance grows over the software evolution. Section 6.3.3 describes the evolution of NOA and NOM proportion in the systems over time.

6.3.1 Evolution of Fan-in and Fan-out Relation

Berard [Berard, 1993] 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. [2007], 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 the complexity and low reusability [Booch, 1991, Berard, 1993, Henderson-Sellers, 1996, Martin, 2003].

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 6.2 summarizes these two types of coupling behavior. The acronyms "UNCP dir." and "NCP dir." refer to "unnecessary coupling direction" and "necessary coupling direction". 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[3].

Table 6.2: Evolution of necessary and unnecessary coupling.

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

Results in Table 6.2 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 carried out a visual analysis of 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 6.1 exhibits the necessary and unnecessary coupling evolution chat in *J2ObjC*. In this case, the unnecessary coupling starts high and decreases over the first versions of the system until the 81^{st} version. After that, it remains stable until the 149^{th} 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 81^{st} version and remains stable for a period. Between the 149^{th} and 152^{th} versions, the unnecessary coupling suffers a slight variation, but it stabilizes again after the 152^{th} 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*,

[3] `https://github.com/BrunoLSousa/SupplementaryMaterialSPEResearch`

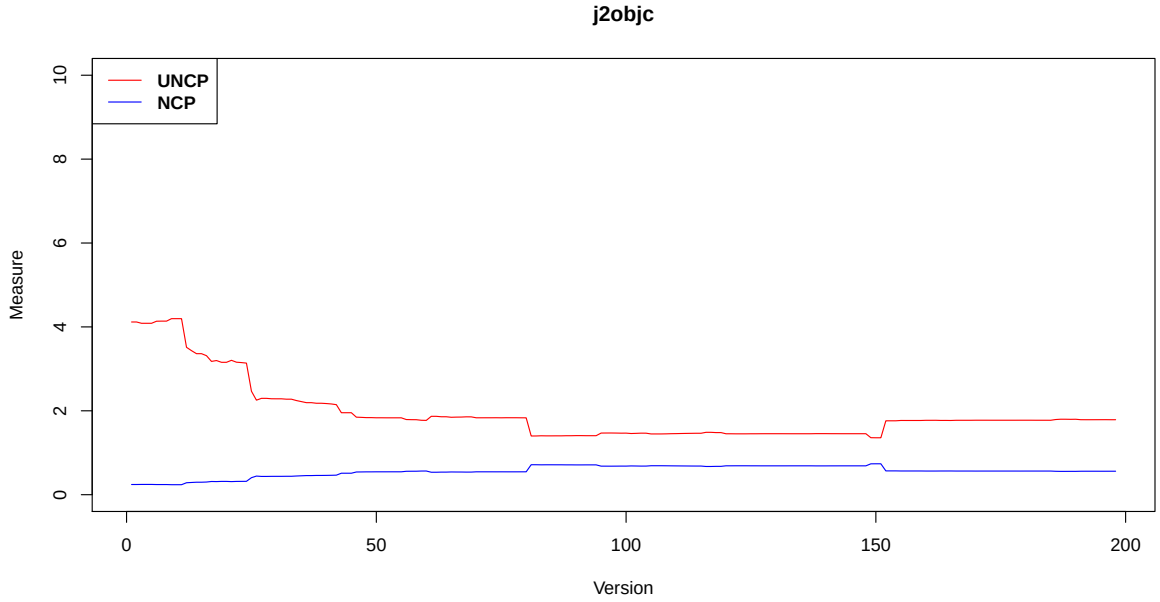

Figure 6.1: Evolution of unnecessary and necessary coupling in *J2ObjC*.

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 significant change. This change was a drop in *Dropwizard*, while it was a climb in *Kafka*. After that, the unnecessary coupling rema stable over the 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 100^{th} 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 100^{th} 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.

6.3.2 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, i.e., classes have been added in the leaves of the tree with minimal impact on DIT. Figure 6.2 shows an example of this behavior.

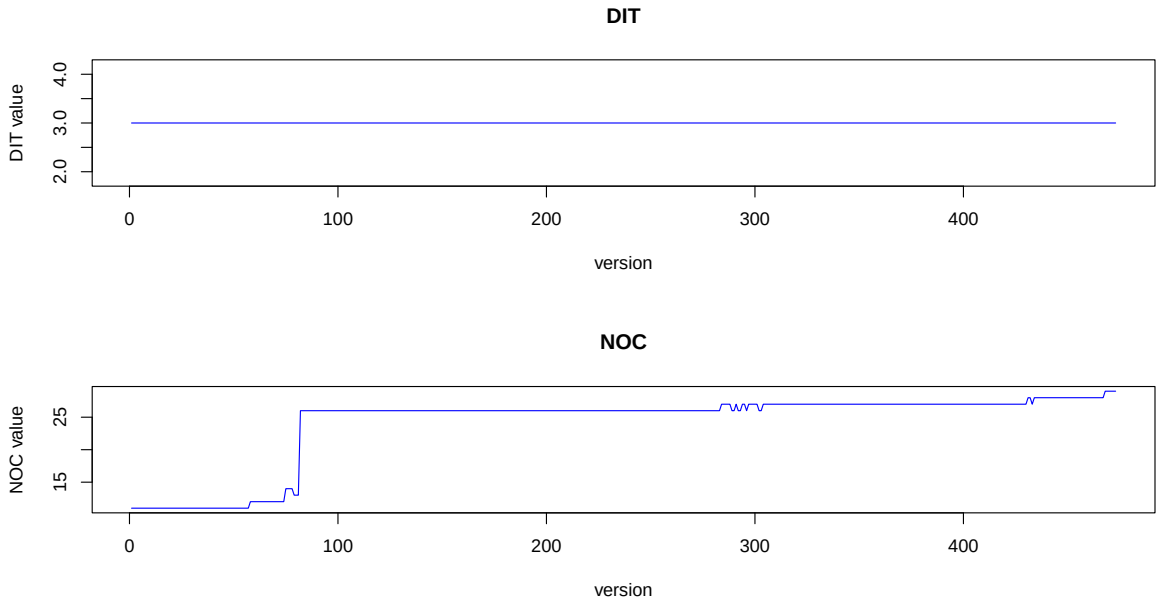

Figure 6.2: Detailing of inheritance hierarchy evolution for a class in *Eclipse JDT Core*.

Figure 6.2 shows the evolution of DIT and NOC metrics in *org.eclipse.jdt.internal.compiler.ast.Expression* inside *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, i.e., classes have been added in the tree's leaves. Such a finding characterizes how this software system's inheritance hierarchy has grown. However, this preliminary analysis needs a deeper investigation to generalize this pattern for all object-oriented Java software systems.

6.3.3 Evolution of NOA and NOM Relation

This section 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 6.3 summarizes the behavior of the proportions. For the cases where the proportions have increased, we put a "+" signal. When the ratios decreased, we included a "-" signal. The charts that show the evolution of the NOA and NOM proportions are available as supplementary material[4].

Table 6.3: Evolution of NOA and NOM proportion.

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

Analyzing Table 6.3, we notice that 35% of the proportions have increased while 65% have decreased. Such observation shows that the number of methods has increased

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

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. *FrameworkBenchmarks* and *Pentaho Kettle* are systems where the proportion exceeded 60%. *FrameworkBenchmarks* 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 6.3 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 Fowler et al. [1999]. 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 47^{th} version, the NOA and NOM proportion decreases and maintains this behavior until it stabilizes close to the 250^{th} version. Many points of the chart in Figure 6.3 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.

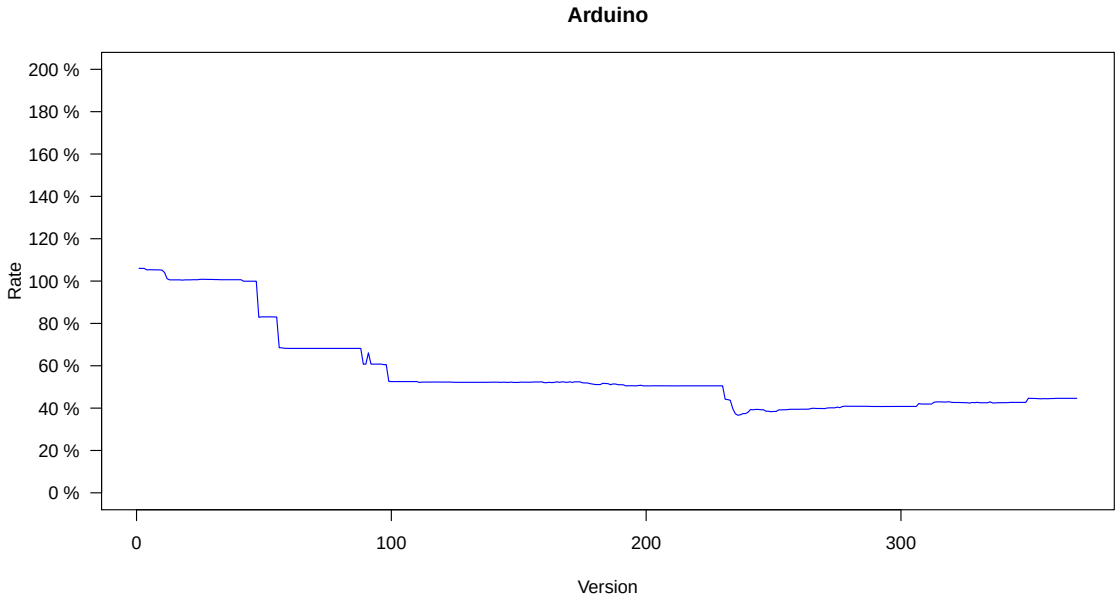

Figure 6.3: Evolution of NOA and NOM proportion in *Arduino*.

6.4 Analysis of Growth and Decrease of Metrics' Values

This section answers **RQ3.** *Which proportion of classes within the software system affects the growth and decrease of internal attributes?*

We analyzed the percentages of classes from the systems directly responsible for increasing and decreasing the metrics' values of the analyzed internal attributes. 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 6.4–6.7. We removed the ghost classes and classes under ten periods from our trend analysis, and we subtracted both of them from the system's whole classes to compute the percentages.

Internal Attributes Growth. Figures 6.4–6.7 show the percentage of classes that present an increasing tendency in their metrics. In this analysis, we considered only the valid classes, i.e., those that are not ghosts and have more than ten periods. 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%, i.e., 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, e.g., refactoring, have contributed

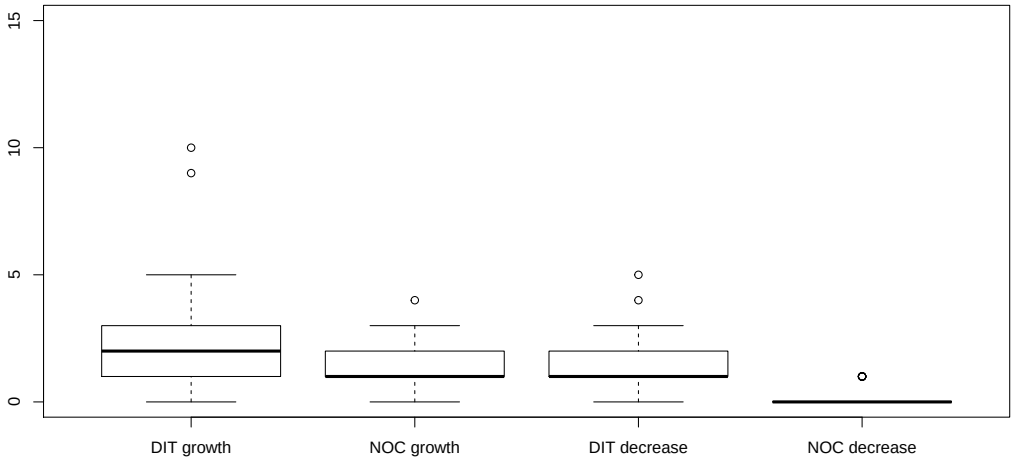

Figure 6.4: Distribution of classes that affect DIT and NOC growth and decrease.

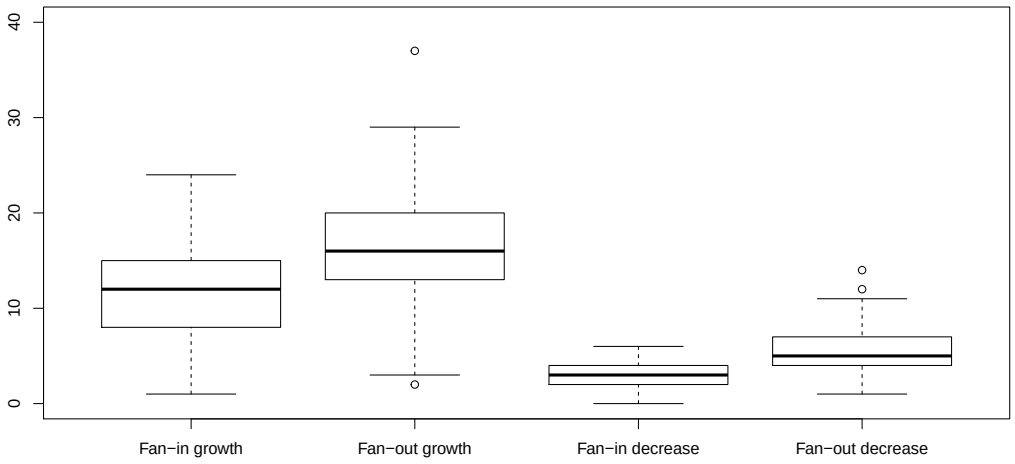

Figure 6.5: Distribution of classes that affect FAN-IN and FAN-OUT growth and decrease.

to not increasing the inheritance metrics' values over the software evolution. In terms of cohesion, no more than 2% of the classes have such metrics increased.

Internal Attributes Decreasing. Figures 6.4–6.7 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, 6%, 5%, 3%, 1%, and 1%, in this sequence.

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

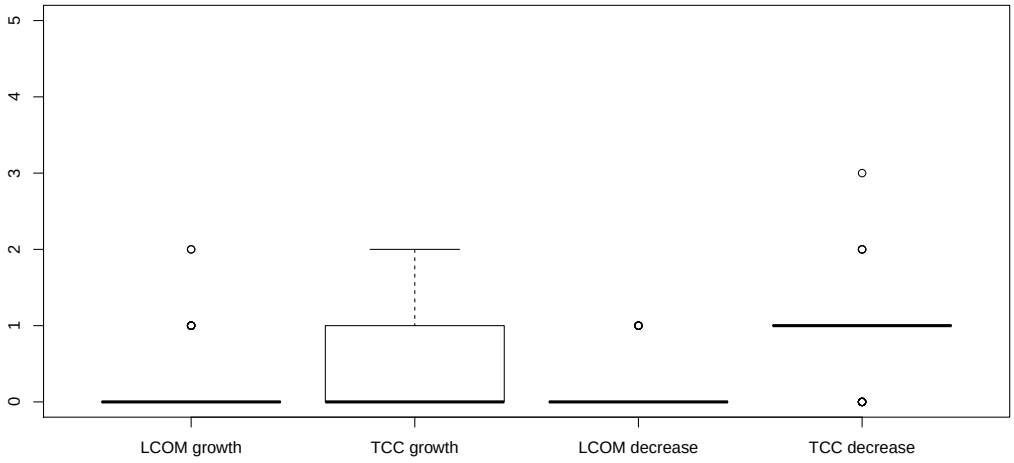

Figure 6.6: Distribution of classes that affect LCOM and TCC growth and decrease.

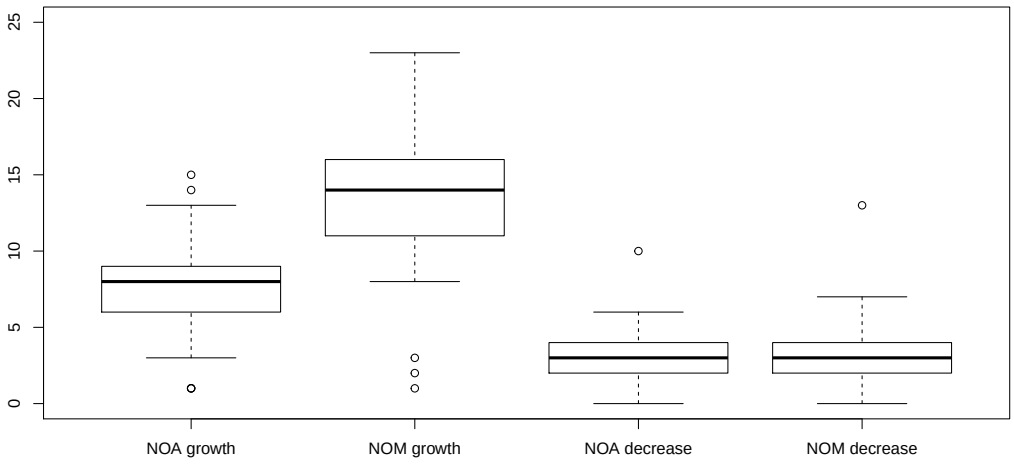

Figure 6.7: Distribution of classes that affect NOA and NOM growth and decrease.

(i) first and second metrics grow,

(ii) first and second metrics decrease,

(iii) the first metric grows while the second metric decreases, and

(iv) the first metric decreases, and the second metric grows.

Table 6.4 summarizes the results obtained for these cases. Each group of four columns contains a pair of metrics separated by a dash, e.g., "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.

Table 6.4: Intersection percentages of the trend results.

System	DIT – NOC				Fan-in – Fan-out				LCOM – TCC				NOA – NOM			
	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%
Framework Benchmarks	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%

The results show that all intersection cases are rare for inheritance hierarchy and cohesion internal attributes. 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 coupling, and *(ii)*, *(iii)* and *(iv)* in class size internal attributes

are rare to occur since they also presented 3% as a maximum percentage. Case *(i)* in both coupling and class size internal attributes 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 of internal attributes 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 attribute does not follow a related pattern.

6.5 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 6.2 by following our evaluation protocol. As stated in Step 2 of Section 5.1, 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 we used the structure and coefficients obtained in the modeling process, using the whole training set, to perform the predictions for the long-term forecast. However, when performing the short-term prediction, we kept the original structure 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 [Bowerman and O'Connell, 1993]. 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:

$$PMSE = \frac{\sum_{i=1}^{n} e_i^2}{n} \tag{6.1}$$

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 large amount of data and extracted many prediction models. However, we made them available online as supplementary material[5].

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. Here, 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 6.8 and 6.9 show the prediction charts we generated after evaluating the data from the phase of test for both short-term and long-term prediction of the model reported here. In both charts, the blue line represents the real value, i.e., 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 6.8, the forecasts were very close to the real values.

For the long-term prediction, in Figure 6.9, the forecasts obtained for the first periods were better, as expected. We can observe that the predicted value was very close

[5] `https://github.com/BrunoLSousa/SupplementaryMaterialSPEResearch`

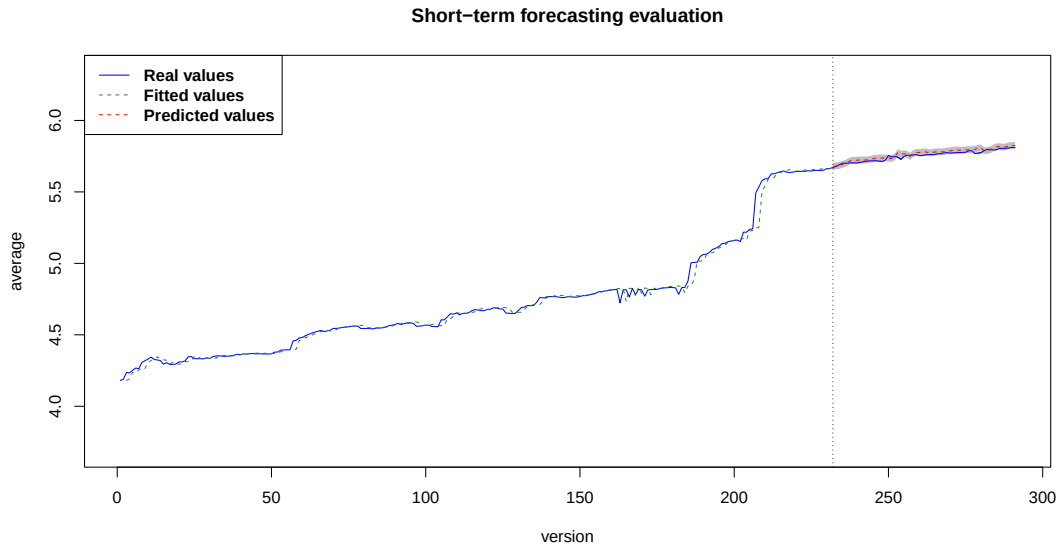

Figure 6.8: Evaluation of the short-term prediction for the model extracted for the *spring-framework* regarding the Fan-out metric.

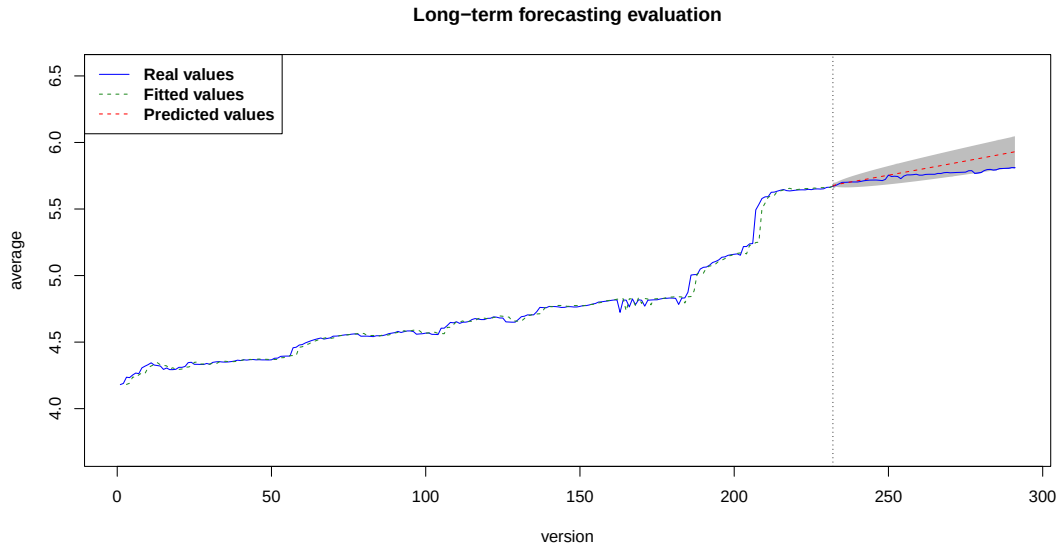

Figure 6.9: Evaluation of the long-term prediction for the model extracted for the *spring-framework* regarding the Fan-out metric.

to the real value in the forecasts performed from the 233^{th} to 260^{th} periods. However, although the forecasts started to lose precision after the 260^{th} period, this behavior is expected because the further the period 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. We compared the PMSE generated from all models described in Section 6.2. Figure 6.10 summarizes the distribution of the errors for the short-term forecasts, and Figure 6.11 shows the distribution of the

PMSE values obtained for the long-term forecasts.

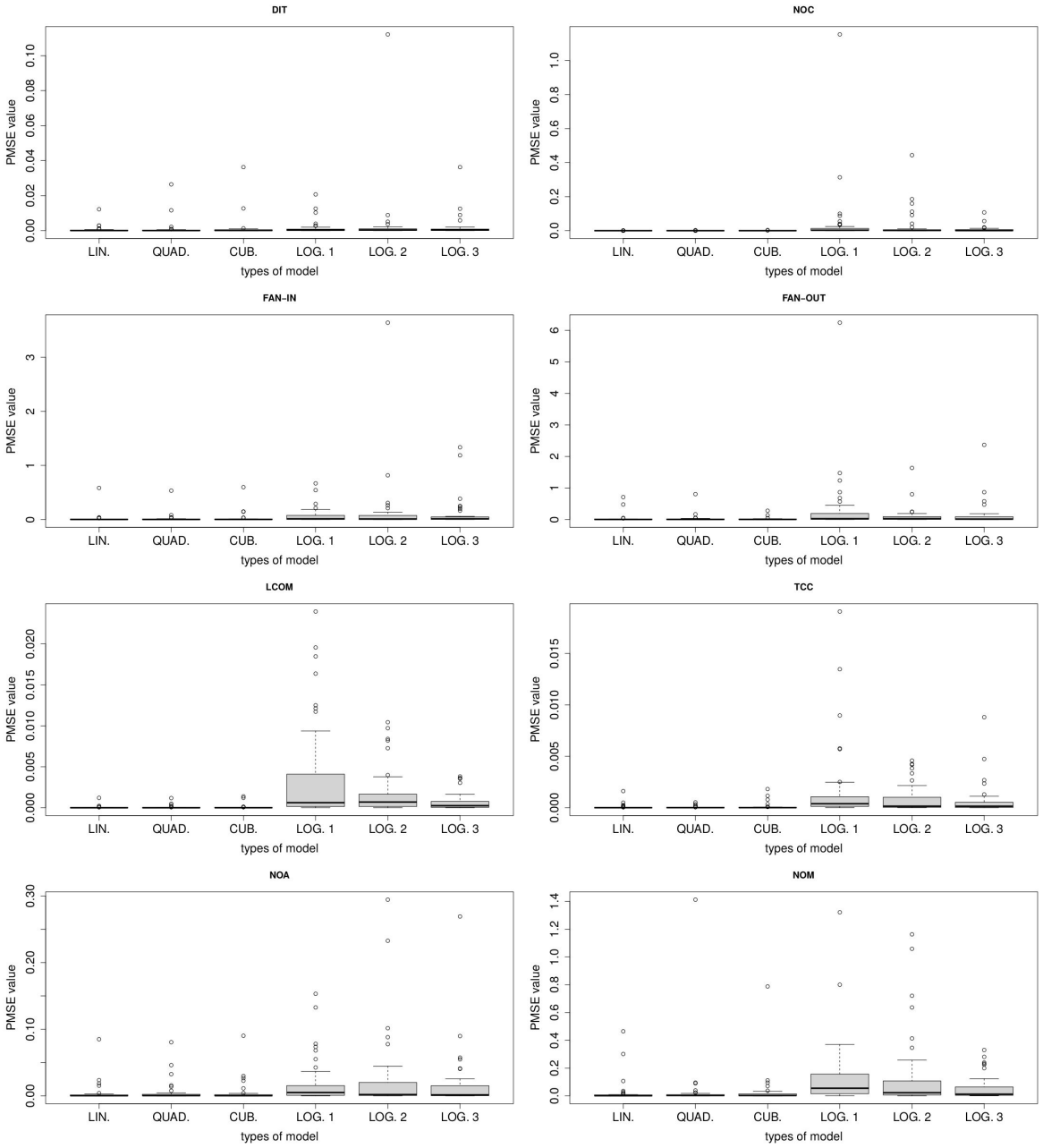

Figure 6.10: Distribution of PMSE of the prediction models extracted for the short-term forecast.

Analyzing Figures 6.10 - 6.11, 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 metrics, we can observe that it has a PMSE close to 0, and it is the one that contains the fewest outliers both for long-term and short-term forecasts. Figure 6.10 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

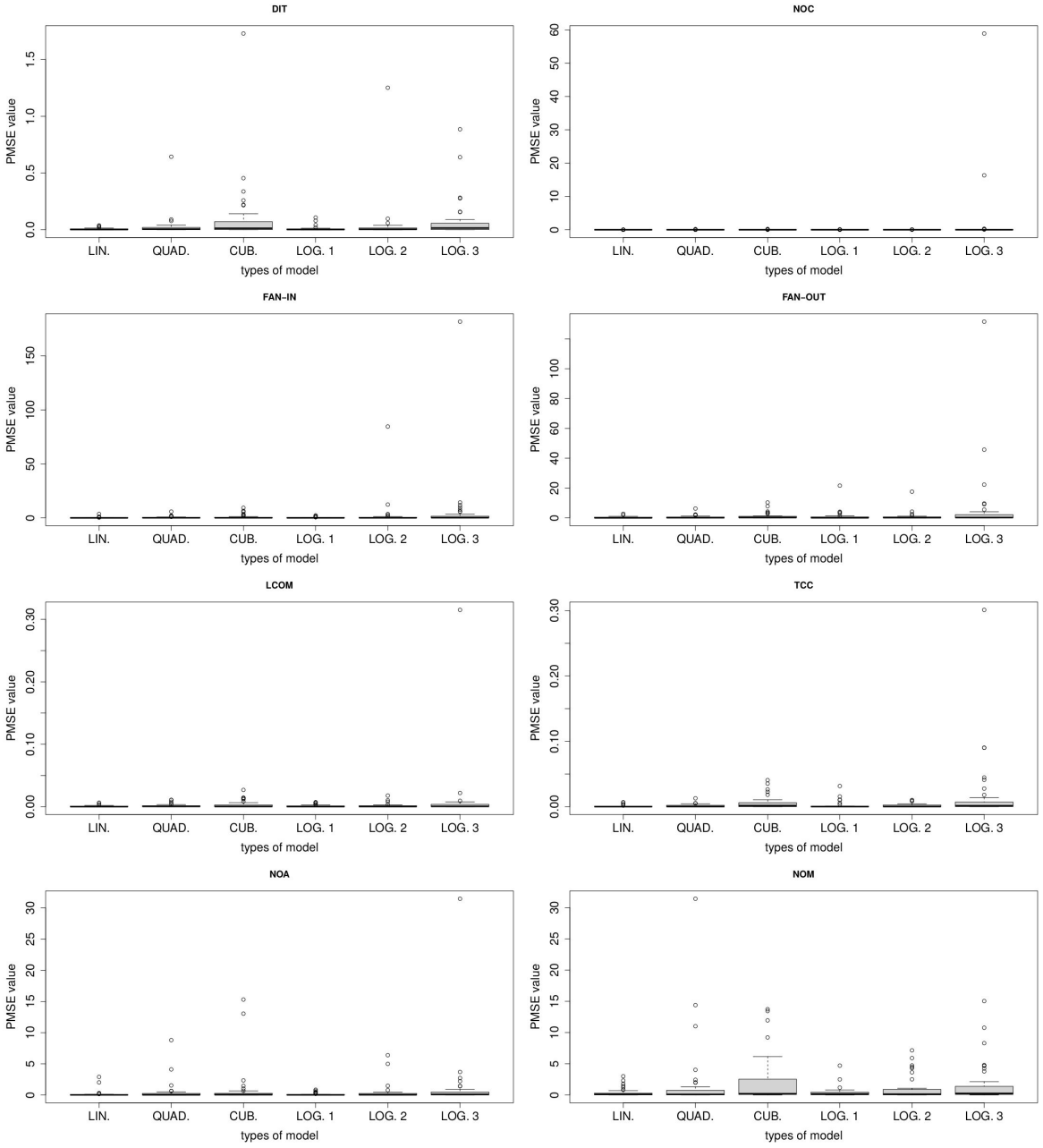

Figure 6.11: Distribution of PMSE of the prediction models extracted for the long-term forecast.

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 6.11, we can see an increase in the PMSE compared with the short-term forecast. However, when analyzing the distribution of the best metrics models, i.e., 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.

6.6 Software Evolution Properties

This section compiles and discusses the results of the empirical analysis carried out in this study. The results lead us to identify ten software evolution properties related to coupling, cohesion, inheritance, and size.

1st - 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.

2nd - 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 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.

3rd - 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. [2015], whose main

conclusion is that bad smells have been introduced in software systems since 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.

4th - 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, i.e., 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, i.e., 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.

5th - 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.

6th - 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.

7th - 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.

8th - 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.

9th - A significant percentage of classes directly contributes to the coupling and size evolution. In contrast, a small portion directly impacts 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, which are probably classes containing business rules of the software system. Such a group represents approximately a quarter of the software system.

10th - 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.

6.7 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. The user collects the necessary data for running the method in this approach. 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[6]. 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[7]. 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

[6] `https://github.com/BrunoLSousa/DSTool`
[7] `https://github.com/BrunoLSousa/TSAnalysisMethod`

of internal aspects of Java open-source systems. They can also help developers to plan strategies for controlling some aspects, e.g., 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 6.5, 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 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.

6.8 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 threaten 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 attributes in 46 open-source Java systems. Although we considered many systems, our results reflect the evolution to only open-source Java systems. We may not generalize our results for systems to other domains and contexts, e.g., proprietary systems or systems written in other object-oriented languages. However, since Java is the most used language in open-source software development, the results presented in this study can 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 [Graves et al., 2000, Ramil and Lehman, 2000, Capiluppi, 2003, Koch, 2005, Arisholm and Briand, 2006, Ratzinger et al., 2007a, Shatnawi and Li, 2008, Kirbas et al., 2014], 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 periods i.e., periods without a measure, making it challenging to fit a model to the data. We applied a reconstruction data approach to ensure we only had time series with valid periods. 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.

6.9 Final Remarks

This chapter presented and discussed an empirical study on object-oriented software systems evolution from the perspective of coupling, cohesion, size, and inheritance hierarchy. Our research considered the dataset we constructed, composed of software metrics time series of 46 Java open-source systems. To study the evolution of these internal attributes, we used fan-in and fan-out, LCOM and TCC, DIT and NOC, and NOA and NOM for representing coupling, cohesion, inheritance hierarchy, and size, respectively. After mapping the internal attributes into software metrics, we applied our two-phase analysis method, defined in Chapter 5, to analyze how these aspects evolve.

We also analyzed the accuracy of our time series approach concerning prediction. According to our analysis, the models extracted by our method accurately predict future values in a short and extended period.

This chapter still discussed the results and observations of the empirical analyses. Based on the results, we identified ten evolution properties for coupling, cohesion, inheritance hierarchy, and size. The set of properties presented in this chapter describes how software evolution occurs in object-oriented software systems from the perspective of these internal attributes.

Chapter 7 concludes this Ph.D. dissertation and indicates future work.

Chapter 7

Conclusion

The evolution process is one of the most critical phases in the software life cycle, comprising about 85% to 90% of the total cost of a software system. Due to the relevance of this phase, studies in the literature have made efforts to investigate this area, aiming to provide novel methods and strategies that aid companies in reducing software costs. Lehman et al. [1997] conducted one of the first studies on this subject. They defined eight laws that describe the evolutionary nature of the software systems and indicate how it occurs. Since then, many works have been done to investigate software evolution. Many have aimed to validate Lehman's laws in various software engineering contexts. Despite the research community's efforts to investigate software evolution, there is still a gap in how the internal structures of object-oriented software systems evolve. The present research, concerned with this gap and wishing to contribute to advancing the state of the art in software evolution, proposes a novel method based on time series to analyze and predict the evolution of internal software quality attributes.

We considered in this work the following internal attributes of internal software quality: coupling, cohesion, inheritance, and size, and applied software metrics to assess these internal attributes.

Initially, we carried out a Systematic Literature Mapping (SLM) to understand the state of the art on software evolution and identify what has been produced regarding models in this topic. Our SLM analysis identified 71 papers published on this subject from 1979 to 2022 and revealed the following as results:

(i) the types of software evolution models that researchers produced;

(ii) the characteristics that have been used as the focus for the proposed models;

(iii) an overview of the type of software evolution datasets existing in the literature;

(iv) the commonly used techniques to build the software evolution models.

In this SLM, we identified research opportunities to cover the need for advances in software evolution. Two of them are covered in this Ph.D. dissertation: the lack of publicly available updated and large software evolution datasets and models for analysis and prediction of internal software quality attributes.

Hence, one of the contributions of this dissertation is a software evolution dataset composed of 46 Java open-source software systems and data regarding 46 software metrics. We have made the dataset publicly available[1]. Making an updated and large dataset publicly available contributes to the state of the art since it may support other researchers to carry out studies on software evolution based on its data.

Besides, we carried out three essential studies in this work. First, we defined a novel method based on time series analysis, linear regression techniques, and trend tests to analyze the evolution of object-oriented systems. The proposal of this method is the main contribution of this Ph.D. dissertation since it contains well-defined steps that can be applied for modeling and analyzing other software internal attributes and increasing the knowledge of how they are evolving.

We analyzed the evolution of object-oriented software systems using our software evolution method to provide a fine-grained view of how software internal structure evolves from the perspective of coupling, inheritance hierarchy, cohesion, and class size. Based on this analysis, we identified ten evolution properties of the analyzed characteristics in object-oriented software systems. Knowing how these characteristics 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.

We also conducted a prediction analysis regarding the software evolution method, showing its efficiency in building prediction models. This analysis is another relevant contribution of this work because it demonstrates that our method is efficient for extracting models that perform accurate predictions for the evolution of internal software quality attributes. Practitioners and researchers can apply our method for monitoring how the internal structure of a software system will be in future releases and then anticipate possible decisions and strategies to reduce the complexity and improve the internal structure quality of the software systems. To aid the application and the replication of this work, we made the scripts we developed and used publicly available[2].

7.1 Future Works

In future works, we suggest the following:

1. Investigate the evolution of other internal characteristics not considered in our work.

2. Replicate the study in other contexts, e.g., for other programming languages.

[1] `https://brunolsousa.github.io/software-evolution-dataset/index.html`
[2] `https://github.com/BrunoLSousa/TSAnalysisMethod`

3. Carry out qualitative analyses to identify the factors that affect the evolution of the internal attributes we considered in this work.

4. Create tools to embed the scripts developed in this dissertation and automate the application of our method for software evolution analysis.

Bibliography

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

Arwa Abuasad and Izzat M Alsmadi. Evaluating the correlation between software defect and design coupling metrics. In 2012 International Conference on Computer, Information and Telecommunication Systems (CITS), pages 1–5. IEEE, 2012.

M. Alenezi and K. Almustafa. Empirical analysis of the complexity evolution in open-source software systems. International Journal of Hybrid Information Technology, 8 (2):257–266, 2015.

M. Alenezi and M. Zarour. Modularity measurement and evolution in object-oriented open-source projects. volume 24-26-September-2015, 2015. doi: 10.1145/2832987.2833013.

Ayman Amin, Lars Grunske, and Alan Colman. An automated approach to forecasting qos attributes based on linear and non-linear time series modeling. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pages 130–139. IEEE, 2012.

Ayman Amin, Lars Grunske, and Alan Colman. An approach to software reliability prediction based on time series modeling. Journal of Systems and Software, 86(7): 1923–1932, 2013.

Mehdi Amoui, Mazeiar Salehie, and Ladan Tahvildari. Temporal software change prediction using neural networks. International Journal of Software Engineering and Knowledge Engineering, 19(07):995–1014, 2009.

M. Aniche. Java code metrics calculator (CK), 2015. Available at: https://github.com/mauricioaniche/ck/. Accessed on January, 2021.

G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo. Modeling clones evolution through time series. In Proceedings IEEE International Conference on Software Maintenance. ICSM 2001, pages 273–280. IEEE, 2001.

E. Arisholm and L.C. Briand. Predicting fault-prone components in a java legacy system. In Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering, pages 8–17. ACM, 2006.

Erik Arisholm, Lionel C Briand, and Audun Foyen. Dynamic coupling measurement for object-oriented software. IEEE Transactions on software engineering, 30(8):491–506, 2004.

L. Aversano, L. Cerulo, and C. Del Grosso. Learning from bug-introducing changes to prevent fault prone code. pages 19–26, 2007. doi: 10.1145/1294948.1294954.

Ankita Bansal. Empirical analysis of search based algorithms to identify change prone classes of open source software. COMPUTER LANGUAGES SYSTEMS & STRUCTURES, 47(2):211–231, JAN 2017.

Holger Bär, Markus Bauer, Oliver Ciupke, Serge Demeyer, Stéphane Ducasse, Michele Lanza, Radu Marinescu, Robb Nebbe, Oscar Nierstrasz, Michael Przybilski, et al. The famoos object-oriented reengineering handbook. SCG FAMOOS, oct, 1999.

E.J. Barry, C.F. Kemerer, and S.A. Slaughter. How software process automation affects software evolution: a longitudinal empirical analysis. Journal of Software Maintenance and Evolution: Research and Practice, 19(1):1–31, 2007.

Edward V. Berard. Essays on Object-oriented Software Engineering (Vol. 1). Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. ISBN 0-13-288895-5.

P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based analysis and prediction for software evolution. In 2012 34th International Conference on Software Engineering (ICSE), pages 419–429, June 2012. doi: 10.1109/ICSE.2012.6227173.

James M. Bieman and Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. In Proceedings of the 1995 Symposium on Software Reusability, SSR '95, page 259–262, New York, NY, USA, 1995. ACM, Association for Computing Machinery. ISBN 0897917391. doi: 10.1145/211782.211856.

Grady Booch. Object Oriented Design with Applications. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991. ISBN 0-8053-0091-0.

S. Bouktif, H. Sahraoui, and F. Ahmed. Predicting stability of open-source software systems using combination of bayesian classifiers. ACM Transactions on Management Information Systems, 5(1), 2014. doi: 10.1145/2555596.

B.L. Bowerman and R.T. O'Connell. Forecasting and Time Series: An Applied Approach. Duxbury classic series. Duxbury Press, 1993. ISBN 9780534932510.

G.E.P. Box and G.M. Jenkins. Time Series Analysis: Forecasting and Control. Holden-Day, San Francisco, 1976.

H. P. Breivold, M. A. Chauhan, and M. A. Babar. A systematic review of studies of open source software evolution. In 2010 Asia Pacific Software Engineering Conference, pages 356–365, Nov 2010. doi: 10.1109/APSEC.2010.48.

J. Businge, A. Serebrenik, and M. van den Brand. An empirical study of the evolution of eclipse third-party plug-ins. In Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), pages 63–72. ACM, 2010.

A. Capiluppi. Models for the evolution of os projects. In International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings., pages 65–74. IEEE, 2003.

A. Capiluppi and J.F. Ramil. Studying the evolution of open source systems at different levels of granularity: Two case studies. pages 113–118, 2004.

A. Capiluppi, M. Morisio, and J.F. Ramil. The evolution of source folder structure in actively evolved open source systems. In 10th International Symposium on Software Metrics, 2004. Proceedings., pages 2–13. IEEE, 2004a.

A. Capiluppi, M. Morisio, and J.F. Ramil. Structural evolution of an open source system: A case study. volume 12, pages 172–182, 2004b.

A. Capiluppi, J. Fernandez-Ramil, J. Higman, H.C. Sharp, and N. Smith. An empirical study of the evolution of an agile-developed software system. In Proceedings of the 29th international conference on Software Engineering, pages 511–518. IEEE Computer Society, 2007.

F. Caprio, G. Casazza, M. Di Penta, and U. Villano. Measuring and predicting the linux kernel evolution. In Proceedings of the International Workshop of Empirical Studies on Software Maintenance. Citeseer, 2001.

Michelle Cartwright. An empirical view of inheritance. Information and Software Technology, 40(14):795–799, 1998.

T. Chaikalis and A. Chatzigeorgiou. Forecasting java software evolution trends employing network models. IEEE Transactions on Software Engineering, 41(6):582–602, 2015. doi: 10.1109/TSE.2014.2381249.

Jitender Kumar Chhabra and Varun Gupta. A survey of dynamic software metrics. Journal of computer science and technology, 25(5):1016–1029, 2010.

S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 20(6):476–493, June 1994. ISSN 0098-5589.

William Jay Conover. Practical nonparametric statistics, volume 350. John Wiley & Sons, 1998.

C. Couto, C. Maffort, R. Garcia, and M. T. Valente. Comets: a dataset for empirical research on software evolution using source code metrics and time series analysis. ACM SIGSOFT Software Engineering Notes, 38(1):1–3, 2013.

C. Couto, P. Pires, M. T. Valente, R. Bigonha, and N. Anquetil. Predicting software defects with causality tests. J. Syst. Softw., 93:24–41, 2014.

C. F. M. Couto. Predicting Software Defects with Causality Tests. PhD thesis, UFMG, Belo Horizonte, Minas Gerais, 2013.

Cesar Couto, Christofer Silva, Marco Tulio Valente, Roberto Bigonha, and Nicolas Anquetil. Uncovering causal relationships between software metrics and bugs. In 2012 16th European Conference on Software Maintenance and Reengineering, pages 223–232. IEEE, 2012.

Paul S. P. Cowpertwait and Andrew V. Metcalfe. Introductory Time Series with R. Springer Publishing Company, Incorporated, 1st edition, 2009. ISBN 0387886974, 9780387886978.

John Daly, Andrew Brooks, James Miller, Marc Roper, and Murray Wood. Evaluating inheritance depth on the maintainability of object-oriented software. Empirical Software Engineering, 1(2):109–132, 1996.

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

D.P. Darcy, S.L. Daniel, and K.J. Stewart. Exploring complexity in open source software: Evolutionary patterns, antecedents, and outcomes. In 2010 43rd Hawaii International Conference on System Sciences, pages 1–11. IEEE, 2010.

N.R. Draper and H. Smith. Applied Regression Analysis. Applied Regression Analysis. Wiley, 1981. ISBN 9780471029953.

Mahmoud O. Elish and Mojeeb Al-Rahman Al-Khiaty. A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software. Journal of Software: Evolution and Process, 25(5):407–437, 2013. doi: https://doi.org/10.1002/smr.1549.

Michael English, Chris Exton, Irene Rigon, and Brendan Cleary. Fault detection and prediction in an open-source software project. In Proceedings of the 5th International Conference on Predictor Models in Software Engineering, PROMISE '09, New York, NY, USA, 2009. Association for Computing Machinery. doi: 10.1145/1540438.1540462.

L. Erlikh. Leveraging legacy system dollars for e-business. IT Professional, 2(3):17–23, 2000.

Sinan Eski and Feza Buzluca. An empirical study on object-oriented metrics and software evolution in order to reduce testing costs by predicting change-prone classes. In 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, pages 566–571. IEEE, 2011.

Zhehao Fan, Zhiyong Feng, Xiao Xue, Shizhan Chen, and Hongyue Wu. Ecosystem evolution analysis and trend prediction of projects in android application framework. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASE '20, page 67–72, New York, NY, USA, 2021. Association for Computing Machinery. doi: 10.1145/3417113.3422185.

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

Kecia Aline Marques Ferreira. Connectivity assessment in object-oriented systems. Master's thesis, Federal University of Minas Gerais, 2006. (In portuguese).

Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

B. Gezici, A. Tarhan, and O. Chouseinoglou. Internal and external quality in the evolution of mobile software: An exploratory study in open-source market. Information and Software Technology, 112:178–200, 2019.

M.W. Godfrey and Q. Tu. Evolution in open source software: A case study. In Proceedings 2000 International Conference on Software Maintenance, pages 131–142. IEEE, 2000.

J.M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J.J. Amor, and D.M. German. Macro-level software evolution: a case study of a large software compilation. Empirical Software Engineering, 14(3):262–285, 2009.

Miguel Goulão, Nelson Fonte, Michel Wermelinger, and Fernando Brito e Abreu. Software evolution prediction using seasonal time analysis: a comparative study. In 2012 16th European Conference on Software Maintenance and Reengineering, pages 213–222. IEEE, 2012.

Georgios Gousios. The ghtorent dataset and tool suite. In Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, page 233–236. IEEE Press, 2013. ISBN 9781467329361.

T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. Predicting fault incidence using software change history. IEEE Transactions on software engineering, 26(7):653–661, 2000.

Tihana Galinac Grbac and Goran Mauša. On the distribution of software faults in evolution of complex systems. In Proceedings of the International Colloquium on Software-Intensive Systems-of-Systems at 10th European Conference on Software Architecture, SiSoS@ECSA '16, New York, NY, USA, 2016. Association for Computing Machinery. doi: 10.1145/3175731.3176181.

F. Grigorio, D. Brito, E. Anjos, and M. Zenha-Rela. On systems project abandonment: An analysis of complexity during development and evolution of floss systems. volume 2015-January, 2015. doi: 10.1109/ICASTECH.2014.7068139.

Neelam Gupta and Praveen Rao. Program execution based module cohesion measurement. In Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001), pages 144–153. IEEE, 2001.

Olle Häggström et al. Finite Markov chains and algorithmic applications, volume 52. Cambridge University Press, 2002.

K.H. Hamed and A.R. Rao. A modified mann-kendall trend test for autocorrelated data. Journal of hydrology, 204(1-4):182–196, 1998.

Rachel Harrison, Steve Counsell, and Reuben Nithi. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. Journal of Systems and Software, 52(2-3):173–179, 2000.

Youssef Hassoun, Roger Johnson, and Steve Counsell. A dynamic runtime coupling metric for meta-level architectures. In Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings., pages 339–346. IEEE, 2004a.

Youssef Hassoun, Roger Johnson, and Steve Counsell. Empirical validation of a dynamic coupling metric. Birkbeck College London, Technical Report BBKCS-04-03, 2004b.

Youssef Hassoun, Steve Counsell, and Roger Johnson. Dynamic coupling metric: proof of concept. IEE Proceedings-Software, 152(6):273–279, 2005.

L. Hatton, D. Spinellis, and M. van Genuchten. The long-term growth rate of evolving software: Empirical results and implications. Journal of Software: Evolution and Process, 29(5):e1847, 2017.

Brian Henderson-Sellers. Object-oriented Metrics: Measures of Complexity. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. ISBN 0-13-239872-9.

I. Herraiz, G. Robles, and J.M. Gonzalez-Barahona. Towards predictor models for large libre software projects. 2005. doi: 10.1145/1083165.1083168.

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

I. Herraiz, J.M. Gonzalez-Barahona, and G. Robles. Towards a theoretical model for software growth. In Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007), pages 21–21. IEEE, 2007a.

Israel Herraiz, Jesus M Gonzalez-Barahona, Gregorio Robles, and Daniel M German. On the prediction of the evolution of libre software projects. In 2007 IEEE International Conference on Software Maintenance, pages 405–414. IEEE, 2007b.

Daniel Honsel, Verena Herbold, Stephan Waack, and Jens Grabowski. Investigation and prediction of open source software evolution using automated parameter mining for agent-based simulation. AUTOMATED SOFTWARE ENGINEERING, 28(1), MAY 2021. doi: 10.1007/s10515-021-00280-3.

V. Honsel, S. Herbold, and J. Grabowski. Hidden markov models for the prediction of developer involvement dynamics and workload. 2016. doi: 10.1145/2972958.2972960.

Cheng Huang, Hui Zhou, Chunyang Ye, and Bingzhuo Li. Code clone detection based on event embedding and event dependency. In Proceedings of the 13th Asia-Pacific Symposium on Internetware, Internetware '22, page 65–74, New York, NY, USA, 2022. Association for Computing Machinery. doi: 10.1145/3545258.3545277.

Ayelet Israeli and Dror G. Feitelson. The linux kernel as a case study in software evolution. Journal of Systems and Software, 83(3):485 – 501, 2010. ISSN 0164-1212.

Maliheh Izadi, Kiana Akbari, and Abbas Heydarnoori. Predicting the objective and priority of issue reports in software repositories. EMPIRICAL SOFTWARE ENGINEERING, 27(2), MAR 2022. doi: 10.1007/s10664-021-10085-3.

C. Izurieta and J. Bieman. The evolution of freebsd and linux. In Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering, pages 204–211. ACM, 2006.

Kirk S. R. Jenkins S. Software architecture graphs as complex networks: A novel partitioning scheme to measure stability and evolution. Information Sciences, 177(12): 2587–2601, 2007.

Maurice George Kendall. Rank correlation methods. Charless Griffin, LDN, 1975.

Benedicte Kenmei, Giuliano Antoniol, and Massimiliano Di Penta. Trend analysis and issue prediction in large-scale open source systems. In 2008 12th European Conference on Software Maintenance and Reengineering, pages 73–82. IEEE, 2008.

Zeinab Azadeh Kermansaravi, Md Saidur Rahman, Foutse Khomh, Fehmi Jaafar, and Yann-Gael Gueheneuc. Investigating design anti-pattern and design pattern mutations and their change- and fault-proneness. EMPIRICAL SOFTWARE ENGINEERING, 26(1), JAN 2021. doi: 10.1007/s10664-020-09900-0.

Taghi M Khoshgoftaar, John C Munson, and David L Lanning. Dynamic system complexity. In [1993] Proceedings First International Software Metrics Symposium, pages 129–140. IEEE, 1993.

Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In Proceedings of the 14th International Conference on Mining Software Repositories, pages 102–112. IEEE press, 2017.

S. Kirbas, A. Sen, B. Caglayan, A. Bener, and R. Mahmutogullari. The effect of evolutionary coupling on software defects: An industrial case study on a legacy system. 2014.

Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007. URL `http://www.dur.ac.uk/ebse/resources/Systematic-reviews-5-8.pdf`.

S. Koch. Evolution of open source software systems–a large-scale investigation. In Proceedings of the 1st International Conference on Open Source Systems, pages 148–153, 2005.

Stefan Koch. Software evolution in open source projects—a large-scale investigation. J. Softw. Maint. Evol.: Res. Pract., 19:361–382, 2007.

S. Krishnan, C. Strasburg, R.R. Lutz, and K. GoÅ¡eva-Popstojanova. Are change metrics good predictors for an evolving software product line? 2011. doi: 10.1145/2020390.2020397.

Y. Lee, J. Yang, and K. H. Chang. Metrics and evolution in open source software. In Seventh International Conference on Quality Software (QSIC 2007), pages 191–197. IEEE, Oct 2007.

Manny M Lehman, Juan F Ramil, and U Sandler. An approach to modelling long-term growth trends in software systems. In Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01), page 219. IEEE Computer Society, 2001.

Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry, and Wladyslaw M Turski. Metrics and laws of software evolution-the nineties view. In Proceedings Fourth International Software Metrics Symposium, pages 20–32. IEEE, 1997.

Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining, pages 177–187, 2005.

D. Li, B. Guo, Y. Shen, J. Li, and Y. Huang. The evolution of open-source mobile applications: An empirical study. Journal of software: Evolution and process, 29(7): e1855, 2017.

Hui Li, Li-Ying Hao, and Rong Chen. Multi-level formation of complex software systems. ENTROPY, 18(5), MAY 2016. doi: 10.3390/e18050178.

Wei Li. Another metric suite for object-oriented programming. J. Syst. Softw., 44(2): 155–162, 1999. ISSN 0164-1212. doi: 10.1016/S0164-1212(98)10052-3.

Bennett P. Lientz and E. Burton Swanson. Software Maintenance Management. Addison-Wesley Longman Publishing Co., Inc., USA, 1980. ISBN 0201042053.

Meili Liu, Xiaogang Qi, and Hao Pan. Multifractal analysis of the software evolution in software networks. CHINESE PHYSICS B, 31(3), FEB 1 2022. doi: 10.1088/1674-1056/ac1b8a.

Mark Lorenz and Jeff Kidd. Object-Oriented Software Metrics: A Practical Guide. Prentice-Hall, Inc., USA, 1994. ISBN 013179292X.

Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki, and Audris Mockus. World of code: an infrastructure for mining the universe of open source vcs data. In 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pages 143–154. IEEE, 2019.

Nisha Malik and Rajender Singh Chhillar. New design metrics for complexity estimation in object oriented systems. International Journal on Computer Science and Engineering, 3(10):3367, 2011.

Maíra Marques, Jocelyn Simmonds, Pedro O. Rossel, and María Cecilia Bastarrica. Software product line evolution: A systematic literature review. Information and Software Technology, 105:190 – 208, 2019. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2018.08.014.

Robert Martin. Oo design quality metrics. An analysis of dependencies, 12(1):151–170, 1994.

Robert Cecil Martin. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003. ISBN 0135974445.

Thomas J McCabe. A complexity measure. IEEE Transactions on software Engineering, (4):308–320, 1976.

T. Mens, J. Fernandez-Ramil, J. Fernandez-Ramil, and S. Degrandsart. The evolution of eclipse. In ICSM, pages 386–395, Sep. 2008.

T. Mens, Y. Guéhéneuc, J. Fernández-Ramil, and M. D'Hondt. Guest editors' introduction: Software evolution. IEEE Software, 27(04):22–25, jul 2010. ISSN 1937-4194. doi: 10.1109/MS.2010.100.

Bertrand Meyer. The many faces of inheritance: A taxonomy of taxonomy. Computer, 29(5):105–108, 1996.

Bertrand Meyer. Object-oriented software construction, volume 2. Prentice hall Englewood Cliffs, 1997.

Jeremy Miles. R Squared, Adjusted R Squared. American Cancer Society, 2014. ISBN 9781118445112.

Deepti Mishra. New inheritance complexity metrics for object-oriented software systems: An evaluation with weyuker's properties. Computing and Informatics, 30(2):267–293, 2012.

A. Mitchell and J. F. Power. Run-time cohesion metrics for the analysis of java programs. Technical Report NUIM-CS-TR-2003-08, National University of Ireland, Kildare, Ireland, 2003.

Aine Mitchell and James F Power. Run-time cohesion metrics: An empirical investigation. In Proc. the International Conference on Software Engineering Research and Practice, pages 532–537, Las Vegas, USA, 2004.

Aine Mitchell and James F Power. Using object-level run-time metrics to study coupling between objects. In Proceedings of the 2005 ACM symposium on Applied computing, pages 1456–1462, 2005.

Áine Mitchell and James F Power. A study of the influence of coverage on the relationship between static and dynamic coupling metrics. Science of Computer Programming, 59 (1-2):4–25, 2006.

Audris Mockus and David M Weiss. Predicting risk of software changes. Bell Labs Technical Journal, 5(2):169–180, 2000.

P.A. Morettin and C.M.C. Toloi. Time Serie Analysis. ABE - Fisher Project. Edgard Blucher, 2006. ISBN 9788521203896. (In portuguese).

John C. Munson and Taghi M. Khoshgoftaar. Measuring dynamic program complexity. IEEE software, 9(6):48–55, 1992.

John C. Munson and Taghi M. Khoshgoftaar. Software Metrics for Reliability Assessment, page 493–529. McGraw-Hill, Inc., USA, 1996. ISBN 0070394008.

C.R. Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. Physical Review E - Statistical, Nonlinear, and Soft Matter Physics, 68(4):046116–1–046116–15, 2003.

Glenford Myers. Reliable Software Through Composite Design. Petrocelli/Charter, 1975. ISBN 0884052842.

Emal Nasseri, Steve Counsell, and M Shepperd. An empirical study of evolution of inheritance in java oss. In 19th Australian Conference on Software Engineering (aswec 2008), pages 269–278. IEEE, 2008.

Mark EJ Newman. The structure and function of complex networks. SIAM review, 45 (2):167–256, 2003.

John T. Nosek and Prashant Palvia. Software maintenance management: Changes in the last decade. Journal of Software Maintenance: Research and Practice, 2(3):157–174, 1990. doi: 10.1002/smr.4360020303.

M.C. Ohlsson, A.A. Andrews, and C. Wohlin. Modelling fault-proneness statistically over a sequence of releases: A case study. Journal of Software Maintenance and Evolution, 13(3):167–199, 2001. doi: 10.1002/smr.229.

W. Pan, B. Li, Y. Ma, and J. Liu. A novel software evolution model based on software networks. Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, 5 LNICST(PART 2):1281–1291, 2009. doi: 10.1007/978-3-642-02469-6_9.

Weifeng Pan, Bing Li, Yutao Ma, and Jing Liu. Multi-granularity evolution analysis of software using complex network theory. Journal of Systems Science and Complexity, 24(6):1068–1082, 2011.

T. Panas, R. Lincke, J. Lundberg, and W. Lowe. A qualitative evaluation of a software development and re-engineering project. In 29th Annual IEEE/NASA Software Engineering Workshop, pages 66–75. IEEE, 2005. doi: 10.1109/SEW.2005.15.

J. Pati, B. Kumar, D. Manjhi, and K.K. Shukla. A comparison among arima, bp-nn, and moga-nn for software clone evolution prediction. IEEE Access, 5:11841–11851, 2017. doi: 10.1109/ACCESS.2017.2707539.

Judea Pearl. Probabilistic reasoning in intelligent systems: networks of plausible inference. Elsevier, 2014.

Jean Petrić and Tihana Galinac Grbac. Software structure evolution and relation to system defectiveness. In Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, page 34. ACM, 2014.

Saheed Popoola, Xin Zhao, Jeff Gray, and Antonio Garcia-Dominguez. Classifying changes to models via changeset metrics. In Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS '22, page 276–285, New York, NY, USA, 2022. Association for Computing Machinery. doi: 10.1145/3550356.3561563.

U. Raja, D.P. Hale, and J.E. Hale. Modeling software evolution defects: A time series approach. Journal of Software Maintenance and Evolution, 21(1):49–71, 2009.

Paul Ralph and Ewan Tempero. Construct validity in software engineering research and software metrics. In Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018, EASE'18, page 13–23, New York, NY, USA, 2018. ACM, Association for Computing Machinery. ISBN 9781450364034. doi: 10.1145/3210459.3210461.

J.F. Ramil and M.M. Lehman. Metrics of software evolution as effort predictors-a case study. In icsm, pages 163–172, 2000.

Ghulam Rasool and Nancy Fazal. Evolution prediction and process support of oss studies: A systematic mapping. Arabian Journal for Science and Engineering, 42(8):3465–3502, 2017.

J. Ratzinger, M. Pinzger, and H. Gall. Eq-mine: Predicting short-term defects for software evolution. volume 4422 LNCS, pages 12 – 26, Braga, Portugal, 2007a.

J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall. Mining software evolution to predict refactoring. pages 354–363, 2007b. doi: 10.1109/ESEM.2007.63.

G. Robles, J.J. Amor, J.M. Gonzalez-Barahona, and I. Herraiz. Evolution and growth in large libre software projects. In Eighth International Workshop on Principles of Software Evolution (IWPSE'05), pages 165–174. IEEE, 2005.

Jukka Ruohonen, Sami Hyrynsalmi, and Ville Leppänen. Software evolution and time series volatility: An empirical exploration. In Proceedings of the 14th International Workshop on Principles of Software Evolution, IWPSE 2015, page 56–65, New York, NY, USA, 2015. Association for Computing Machinery. doi: 10.1145/2804360.2804367.

SEDataset. A time series-based dataset of open-source software evolution. `https://brunolsousa.github.io/software-evolution-dataset/index.html`, 2021.

H. Shariat Yazdi, L. Angelis, T. Kehrer, and U. Kelter. A framework for capturing, statistically modeling and analyzing the evolution of software models. Journal of Systems and Software, 118:176–207, 2016. doi: 10.1016/j.jss.2016.05.010.

R. Shatnawi and W. Li. The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. Journal of systems and software, 81 (11):1868–1882, 2008.

RogéRio De Carvalho Silva, Paulo Roberto Farah, and Silvia Regina Vergilio. Machine learning for change-prone class prediction: A history-based approach. In Proceedings of the XXXVI Brazilian Symposium on Software Engineering, SBES '22, page 289–298, New York, NY, USA, 2022. Association for Computing Machinery. doi: 10.1145/3555228.3555249.

G. Singh and M.D. Ahmed. Effect of coupling on change in open source java systems. 2017. doi: 10.1145/3014812.3014835.

Ian Sommerville. Software Engineering. Pearson, 9th edition, Jan 2012.

Bruno L Sousa, Mariza AS Bigonha, and Kecia AM Ferreira. Analysis of coupling evolution on open source systems. In Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse, pages 23–32, 2019a.

Bruno L. Sousa, Mívian M. Ferreira, Kecia A. M. Ferreira, and Mariza A. S. Bigonha. Software engineering evolution: The history told by icse. In Proceedings of the XXXIII Brazilian Symposium on Software Engineering, page 17–21, New York, NY, USA, 2019b. Association for Computing Machinery. ISBN 9781450376518. doi: 10.1145/3350768.3350794.

Bruno Luan de Sousa, Mariza Andrade da Silva Bigonha, Kecia Aline Marques Ferreira, and Glaura da Conceição Franco. Characterizing the evolution of size and inheritance in object-oriented software. In XX Brazilian Symposium on Software Quality, SBQS '21, New York, NY, USA, 2021. Association for Computing Machinery. doi: 10.1145/3493244.3493247.

MM Mahbubul Syeed, Imed Hammouda, and Tarja Systä. Evolution of open source software projects: A systematic literature review. JSW, 8(11):2815–2829, 2013.

Ewan Tempero, James Noble, and Hayden Melton. How do java programs use inheritance? an empirical study of inheritance in java software. In European Conference on Object-Oriented Programming, pages 667–691. Springer, 2008.

Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The qualitas corpus: A curated collection of java code for empirical studies. In APSEC, editor, Software Engineering Conference (APSEC), 2010 17th Asia Pacific, pages 336–345. IEEE, 2010.

R. Trindade, T. Orfanó, K. Ferreira, and E. Wanner. The dance of classes - a stochastic model for software structure evolution. In WETSoM, pages 22–28, May 2017.

Dimitrios Tsoukalas, Marija Jankovic, Miltiadis Siavvas, Dionysios Kehagias, Alexander Chatzigeorgiou, and Dimitrios Tzovaras. On the applicability of time series models for technical debt forecasting. In 15th China-Europe International Symposium on software engineering education, 2019.

Dimitrios Tsoukalas, Dionysios Kehagias, Miltiadis Siavvas, and Alexander Chatzigeorgiou. Technical debt forecasting: an empirical study on open-source repositories. Journal of Systems and Software, 170:110777, 2020.

Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, page 403–414. IEEE Press, 2015. ISBN 9781479919345.

Charles D. Tupper. Object and object/relational databases. In Charles D. Tupper, editor, Data Architecture, pages 369–383. Morgan Kaufmann, Boston, 2011. ISBN 978-0-12-385126-0. doi: https://doi.org/10.1016/B978-0-12-385126-0.00021-8.

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

Stéphane Vaucher and Houari Sahraoui. Do software libraries evolve differently than applications? an empirical investigation. In Proceedings of the 2007 Symposium on Library-Centric Software Design, LCSD '07, page 88–96, New York, NY, USA, 2007. Association for Computing Machinery. doi: 10.1145/1512762.1512771.

Birgit Vogel-Heuser, Alexander Fay, Ina Schaefer, and Matthias Tichy. Evolution of software in automated production systems: Challenges and research directions. Journal of Systems and Software, 110:54–84, 2015.

Ana Vrankovic, Tihana Galinac Grbac, and Zeljka Car. Software structure evolution and relation to subgraph defectiveness. IET SOFTWARE, 13(5):355–367, OCT 2019. doi: 10.1049/iet-sen.2018.5060.

Vijay Walunj, Gharib Gharibi, Duy H. Ho, and Yugyung Lee. Graphevo: Characterizing and understanding software evolution using call graphs. In 2019 IEEE

International Conference on Big Data (Big Data), pages 4799–4807, 2019. doi: 10.1109/BigData47090.2019.9005560.

Vijay Walunj, Gharib Gharibi, Rakan Alanazi, and Yugyung Lee. Defect prediction using deep learning with network portrait divergence for software evolution. EMPIRICAL SOFTWARE ENGINEERING, 27(5), SEP 2022. doi: 10.1007/s10664-022-10147-0.

L. Wang, Z. Wang, C. Yang, L. Zhang, and Q. Ye. Linux kernels as complex networks: A novel method to study evolution. pages 41–50, 2009. doi: 10.1109/ICSM.2009.5306348.

L. Wang, Z. Wang, C. Yang, and L. Zhang. Evolution and stability of linux kernels based on complex networks. Science China Information Sciences, 55(9):1972–1982, 2012. doi: 10.1007/s11432-011-4337-1.

W.W.S. Wei. Time Series Analysis: Univariate and Multivariate Methods. Pearson Addison Wesley, 2006. ISBN 9780321322166.

Mark Weiser. Program slicing. IEEE Transactions on software engineering, (4):352–357, 1984.

Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In 18th EASE, pages 1–10, 2014.

C Murray Woodside. A mathematical model for the evolution of software. Journal of Systems and Software, 1:337–345, 1979.

Wenjin Wu, Wen Zhang, Ye Yang, and Qing Wang. Time series analysis for bug number prediction. In The 2nd International Conference on Software Engineering and Data Mining, pages 589–596. IEEE, 2010.

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

Sherif M Yacoub, Hany H Ammar, and Tom Robinson. Dynamic metrics for object oriented designs. In Proceedings Sixth International Software Metrics Symposium (Cat. No. PR00403), pages 50–61. IEEE, 1999.

Xiaoxing Yang, Ke Tang, and Xin Yao. A learning-to-rank approach to software defect prediction. IEEE Transactions on Reliability, 64(1):234–246, 2014.

H.S. Yazdi, M. Mirbolouki, P. Pietsch, T. Kehrer, and U. Kelter. Analysis and prediction of design model evolution using time series. In International Conference on Advanced Information Systems Engineering, pages 1–15. Springer, 2014.

J. Zhang, S. Sagar, and E. Shihab. The evolution of mobile apps: An exploratory study. In Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile, pages 1–8. ACM, 2013.

Xiaolong Zheng, Daniel Zeng, Huiqian Li, and Feiyue Wang. Analyzing open-source software systems as complex networks. Physica A: Statistical Mechanics and its Applications, 387(24):6190–6200, 2008.

Liang Zong. Classification based software defect prediction model for finance software system - an industry study. In Proceedings of the 2019 3rd International Conference on Software and E-Business, ICSEB 2019, page 60–65, New York, NY, USA, 2020. Association for Computing Machinery. doi: 10.1145/3374549.3374553.

Ioannis Zozas, Stamatia Bibi, and Apostolos Ampatzoglou. Forecasting the principal of code technical debt in javascript applications. IEEE Transactions on Software Engineering, pages 1–15, 2022. doi: 10.1109/TSE.2022.3222318.

Appendix A

Results of the Internal Attributes Modeling

This chapter presents the results obtained for RQ1 after applying the first phase of our time series-based method for modeling the global time series of the internal attributes software metrics. We summarized the results in tables A.1 - A.8. We used some colors to highlight the steps followed in the protocol of our method. We describe the used colors and their meaning as follows.

- **White:** indicates models that do not have a good fit;
- **Green:** indicates models with $\overline{R}^2$ more than 80%;
- **Yellow:** indicates the type(s) of the model that better modeled most of the analyzed systems for that metric;
- **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.

A.1 Results of the DIT modeling

This section shows the results obtained for DIT modeling in Table A.1.

A.2 Results of the NOC modeling

This section shows the results obtained for NOC modeling in Table A.2.

Table A.1: $\overline{R}^2$ values computed from the DIT models.

System	lin.	quad.	cub.	log. 1	log. 2	log. 3
Alluxio	81.34%	83.42%	-	81.05%	83.16%	-
Antlr4	97.27%	97.26%	97.33%	97.23%	97.22%	97.29%
Arduino	93.01%	92.71%	89.94%	92.70%	92.49%	89.59%
Bazel	97.70%	97.70%	97.78%	97.64%	97.64%	97.72%
Bisq	92.48%	91.91%	92.64%	93.24%	93.31%	93.36%
Buck	90.88%	90.81%	91.19%	91.04%	90.98%	91.33%
CAS	97.23%	97.33%	97.14%	97.01%	97.12%	-
CoreNLP	92.50%	92.88%	93.38%	92.41%	92.80%	93.30%
Dbeaver	98.63%	96.97%	98.67%	98.59%	96.88%	98.62%
Dropwizard	82.61%	82.52%	80.24%	82.53%	82.44%	82.41%
Druid	90.52%	89.82%	90.49%	90.19%	89.49%	90.15%
Eclipse JDT Core	98.89%	98.88%	98.88%	99.05%	99.04%	99.04%
Eclipse PDE UI	73.37%	72.61%	73.53%	65.55%	64.50%	65.31%
Elasticsearch	89.36%	-	88.60%	90.05%	89.88%	-
Equinox Framework	-	83.20%	-	-	83.40%	-
FrameworkBenchmarks	97.55%	97.64%	97.76%	97.58%	97.67%	97.78%
Gocd	75.52%	-	-	74.92%	73.45%	-
Graylog	75.77%	75.29%	75.96%	75.62%	75.08%	75.74%
Guava	95.62%	95.19%	95.62%	95.76%	95.36%	95.76%
Hibernate Orm	92.42%	91.62%	92.48%	92.89%	92.10%	92.94%
J2ObjC	98.91%	98.90%	98.90%	99.01%	99.01%	99.01%
Jabref	98.97%	98.96%	98.98%	98.98%	98.98%	98.89%
Jenkins	97.24%	97.17%	97.28%	97.23%	97.16%	97.26%
Jitsi	45.59%	-	43.14%	42.15%	-	43.02%
JMeter	95.54%	95.09%	96.62%	95.42%	95.01%	96.51%
JUnit 5	98.33%	97.78%	98.32%	98.27%	97.72%	98.26%
K-9 Mail	92.49%	91.85%	92.70%	93.05%	92.66%	93.26%
Kafka	85.80%	-	-	85.84%	-	-
LanguageTool	83.96%	93.82%	86.61%	83.58%	93.96%	86.27%
Lucene	64.83%	66.31%	68.46%	73.76%	74.12%	77.16%
MinecraftForge	60.19%	56.94%	55.22%	55.86%	52.75%	50.76%
Neo4j	97.91%	97.93%	97.29%	97.87%	97.88%	97.29%
Netty	97.37%	95.33%	96.48%	97.29%	95.36%	96.48%
OpenRefine	79.03%	79.20%	79.64%	77.49%	77.58%	77.98%
OrientDB	80.69%	79.63%	80.96%	80.83%	79.77%	81.10%
Pentaho Kettle	99.05%	99.07%	99.09%	99.06%	99.10%	99.10%
Pentaho Platform	95.84%	95.88%	95.86%	95.87%	95.91%	95.88%
Pinpoint	96.52%	96.40%	96.57%	96.43%	96.42%	96.48%
PMD	97.13%	-	97.26%	97.61%	-	97.71%
Realm Java	95.63%	95.79%	95.72%	95.58%	95.75%	95.67%
RxJava	84.82%	85.49%	-	84.96%	85.83%	-
Spring Boot	87.28%	88.95%	85.72%	87.14%	88.83%	85.68%
Spring Framework	99.77%	99.77%	99.77%	99.77%	99.78%	99.78%
Spring Security	98.95%	98.95%	99.43%	98.99%	98.99%	99.44%
Tomcat	99.50%	99.51%	99.15%	99.49%	99.49%	99.14%
Tutorials	89.77%	92.11%	-	89.84%	92.20%	-

A.3 Results of the Fan-in modeling

This section shows the results obtained for Fan-in modeling in Table A.3.

Table A.2: $\overline{R}^2$ values computed from the NOC models.

System	lin.	quad.	cub.	log. 1	log. 2	log. 3
Alluxio	88.88%	87.34%	89.40%	90.43%	87.46%	90.63%
Antlr4	95.11%	95.31%	95.38%	93.25%	94.16%	94.54%
Arduino	94.85%	94.70%	94.67%	95.84%	96.44%	96.06%
Bazel	96.71%	96.82%	-	96.73%	96.85%	-
Bisq	91.79%	90.53%	-	90.24%	90.16%	90.66%
Buck	83.19%	83.80%	84.15%	86.10%	86.70%	86.95%
CAS	99.00%	99.03%	98.99%	98.86%	98.88%	98.89%
CoreNLP	94.46%	94.45%	94.51%	94.73%	94.71%	94.78%
Dbeaver	99.61%	99.62%	99.63%	99.57%	99.58%	99.59%
Dropwizard	87.59%	83.96%	87.86%	85.32%	82.05%	85.79%
Druid	91.19%	91.25%	90.31%	89.98%	90.01%	89.01%
Eclipse JDT Core	97.59%	97.59%	97.58%	97.39%	97.53%	97.53%
Eclipse PDE UI	64.85%	95.37%	95.42%	98.73%	98.74%	98.75%
Elasticsearch	65.41%	65.37%	67.06%	64.73%	63.75%	65.79%
Equinox Framework	93.70%	93.93%	-	94.40%	94.57%	-
FrameworkBenchmarks	98.78%	98.77%	-	98.88%	98.60%	-
Gocd	81.78%	82.39%	-	80.39%	80.74%	81.73%
Graylog	97.23%	96.02%	96.79%	96.85%	96.87%	98.99%
Guava	94.68%	94.80%	94.91%	94.40%	94.53%	94.63%
Hibernate Orm	65.46%	66.33%	63.40%	72.58%	73.13%	71.79%
J2ObjC	82.44%	82.69%	83.02%	84.55%	84.56%	84.78%
Jabref	99.35%	99.36%	99.22%	99.30%	99.30%	99.31%
Jenkins	98.85%	98.78%	98.87%	98.84%	98.77%	98.87%
Jitsi	80.66%	91.07%	82.78%	80.58%	90.99%	82.71%
JMeter	-	73.13%	74.14%	-	72.86%	73.79%
JUnit 5	99.28%	99.28%	99.29%	99.37%	99.37%	99.37%
K-9 Mail	97.31%	97.33%	96.29%	97.55%	97.59%	-
Kafka	82.45%	-	-	82.43%	-	-
LanguageTool	93.31%	93.44%	93.50%	93.24%	93.36%	93.41%
Lucene	60.50%	62.62%	65.20%	79.09%	79.15%	79.94%
MinecraftForge	72.85%	68.31%	69.88%	47.60%	48.08%	-
Neo4j	94.46%	94.64%	-	94.81%	94.94%	94.75%
Netty	85.90%	81.86%	77.41%	85.59%	81.72%	77.04%
OpenRefine	87.50%	88.28%	88.36%	87.40%	87.84%	87.92%
OrientDB	94.98%	95.35%	95.28%	94.87%	95.29%	95.22%
Pentaho Kettle	99.47%	99.50%	99.49%	99.50%	99.54%	99.53%
Pentaho Platform	93.00%	93.42%	93.43%	94.38%	94.67%	-
Pinpoint	98.28%	98.28%	98.30%	98.12%	98.00%	98.15%
PMD	89.28%	89.35%	86.67%	87.24%	87.28%	84.25%
Realm Java	93.74%	94.06%	93.83%	93.77%	94.12%	93.07%
RxJava	92.34%	91.88%	92.38%	88.76%	88.61%	89.24%
Spring Boot	90.33%	90.97%	91.20%	89.76%	90.32%	90.75%
Spring Framework	99.67%	99.69%	99.66%	99.66%	99.68%	99.65%
Spring Security	97.80%	98.08%	98.08%	97.93%	98.27%	98.26%
Tomcat	99.29%	98.33%	98.98%	99.24%	98.21%	98.94%
Tutorials	93.27%	93.22%	93.79%	93.45%	93.40%	93.86%

A.4 Results of the Fan-out modeling

This section shows the results obtained for Fan-out modeling in Table A.4.

Table A.3: $\overline{R}^2$ values computed from the Fan-in models.

System	lin.	quad.	cub.	log. 1	log. 2	log. 3
Alluxio	92.73%	91.43%	-	92.80%	92.88%	-
Antlr4	98.29%	98.42%	98.43%	98.53%	98.59%	98.61%
Arduino	94.90%	94.94%	94.94%	90.44%	90.41%	90.38%
Bazel	97.55%	-	96.21%	97.50%	-	96.04%
Bisq	98.26%	98.33%	98.40%	97.50%	97.61%	97.83%
Buck	94.29%	93.83%	94.22%	94.21%	94.17%	94.13%
CAS	99.24%	99.24%	99.26%	99.38%	99.35%	99.40%
CoreNLP	97.45%	97.49%	97.50%	97.48%	97.52%	97.53%
Dbeaver	99.44%	99.46%	99.47%	99.43%	99.44%	99.45%
Dropwizard	92.80%	93.05%	98.24%	93.37%	93.56%	98.48%
Druid	97.63%	97.62%	97.65%	97.23%	97.22%	97.27%
Eclipse JDT Core	98.00%	98.28%	98.28%	98.13%	98.40%	98.39%
Eclipse PDE UI	63.68%	66.00%	62.99%	37.91%	42.59%	35.55%
Elasticsearch	84.87%	84.84%	86.18%	85.64%	85.59%	86.32%
Equinox Framework	98.73%	98.48%	98.83%	98.72%	98.43%	98.78%
FrameworkBenchmarks	99.40%	99.34%	99.39%	99.42%	99.42%	99.41%
Gocd	49.18%	49.50%	54.14%	-	58.47%	60.58%
Graylog	81.11%	81.02%	80.95%	80.41%	80.32%	80.26%
Guava	96.61%	96.61%	96.67%	96.61%	96.61%	96.68%
Hibernate Orm	97.77%	97.89%	97.75%	98.15%	98.44%	98.40%
J2ObjC	94.08%	93.97%	94.29%	92.62%	92.23%	92.91%
Jabref	96.99%	96.99%	97.03%	96.90%	96.90%	96.87%
Jenkins	95.32%	95.37%	95.44%	95.42%	95.46%	95.53%
Jitsi	97.63%	97.96%	97.10%	97.61%	97.96%	97.12%
JMeter	74.39%	76.44%	-	74.38%	76.42%	-
JUnit 5	98.33%	98.33%	98.38%	98.28%	98.28%	98.33%
K-9 Mail	98.60%	98.29%	98.29%	98.16%	98.16%	98.15%
Kafka	-	78.31%	78.02%	-	78.19%	77.90%
LanguageTool	75.27%	75.90%	-	75.24%	75.87%	-
Lucene	45.36%	-	-	37.58%	-	-
MinecraftForge	52.39%	52.46%	48.53%	43.29%	43.54%	44.06%
Neo4j	77.30%	77.25%	-	77.12%	77.07%	-
Netty	75.65%	-	-	75.05%	-	-
OpenRefine	82.58%	83.44%	83.36%	85.50%	86.48%	85.88%
OrientDB	97.58%	97.60%	97.59%	97.46%	97.49%	97.48%
Pentaho Kettle	95.73%	95.72%	95.72%	95.40%	95.39%	95.39%
Pentaho Platform	84.10%	-	84.99%	81.04%	-	79.83%
Pinpoint	96.01%	95.44%	96.21%	96.12%	95.35%	96.33%
PMD	90.96%	90.90%	89.96%	93.10%	93.06%	93.41%
Realm Java	93.71%	94.45%	94.67%	94.05%	94.77%	95.02%
RxJava	79.03%	79.71%	80.63%	78.18%	78.66%	79.15%
Spring Boot	94.18%	94.11%	94.24%	93.74%	93.68%	93.86%
Spring Framework	99.50%	99.51%	99.51%	99.48%	99.48%	99.48%
Spring Security	88.74%	88.73%	87.86%	87.56%	87.56%	86.80%
Tomcat	97.53%	97.55%	97.55%	97.56%	97.58%	97.59%
Tutorials	94.41%	94.84%	95.31%	95.17%	95.51%	96.04%

A.5 Results of the LCOM modeling

This section shows the results obtained for LCOM modeling in Table A.4.

Table A.4: $\overline{R}^2$ values computed from the Fan-out models.

System	lin.	quad.	cub.	log. 1	log. 2	log. 3
Alluxio	91.56%	92.55%	-	91.69%	92.64%	-
Antlr4	75.24%	73.77%	75.34%	74.63%	72.83%	74.73%
Arduino	82.64%	80.93%	81.72%	74.77%	73.36%	72.79%
Bazel	99.35%	99.34%	98.85%	99.33%	99.33%	98.81%
Bisq	97.04%	97.13%	96.84%	96.38%	96.53%	96.33%
Buck	98.99%	99.01%	99.01%	99.06%	99.08%	99.08%
CAS	99.77%	99.77%	99.78%	99.83%	99.83%	99.83%
CoreNLP	98.65%	98.72%	-	98.57%	98.66%	98.58%
Dbeaver	99.58%	99.59%	99.60%	99.58%	99.59%	99.60%
Dropwizard	94.77%	94.77%	94.88%	94.71%	94.71%	94.82%
Druid	97.63%	97.62%	97.65%	97.32%	97.31%	97.35%
Eclipse JDT Core	98.47%	98.47%	98.47%	98.74%	98.74%	98.73%
Eclipse PDE UI	87.19%	87.59%	86.97%	86.37%	86.84%	85.67%
Elasticsearch	72.25%	-	78.17%	72.14%	-	77.66%
Equinox Framework	96.66%	96.71%	96.49%	96.56%	96.60%	96.38%
FrameworkBenchmarks	99.23%	99.13%	99.23%	99.25%	99.16%	99.24%
Gocd	77.22%	78.52%	79.77%	70.77%	67.25%	72.26%
Graylog	85.98%	86.67%	86.68%	84.94%	83.63%	83.61%
Guava	92.94%	92.93%	92.91%	92.92%	92.91%	92.89%
Hibernate Orm	85.71%	81.70%	86.63%	86.71%	82.73%	87.57%
J2ObjC	97.59%	97.62%	97.69%	96.43%	96.39%	96.61%
Jabref	98.06%	98.05%	98.20%	98.06%	98.06%	98.26%
Jenkins	94.10%	94.29%	94.42%	94.22%	94.41%	94.54%
Jitsi	97.91%	-	97.93%	97.95%	-	97.97%
JMeter	83.03%	87.56%	90.71%	83.39%	88.08%	90.90%
JUnit 5	97.69%	97.68%	97.72%	97.42%	97.41%	97.46%
K-9 Mail	93.65%	94.34%	93.99%	93.32%	93.67%	93.18%
Kafka	88.09%	-	90.76%	88.03%	-	90.66%
LanguageTool	71.71%	71.26%	71.70%	70.79%	70.31%	70.39%
Lucene	55.65%	-	48.20%	31.79%	-	-
MinecraftForge	51.59%	53.70%	49.97%	39.64%	40.19%	40.45%
Neo4j	88.66%	88.61%	-	88.86%	88.81%	-
Netty	67.52%	-	67.54%	66.23%	-	66.91%
OpenRefine	37.09%	40.22%	-	27.27%	-	-
OrientDB	98.34%	98.34%	98.31%	98.25%	98.25%	98.22%
Pentaho Kettle	92.89%	92.85%	92.93%	92.86%	92.82%	92.90%
Pentaho Platform	87.89%	88.46%	-	86.96%	87.52%	-
Pinpoint	98.74%	98.66%	98.76%	98.83%	98.78%	98.90%
PMD	95.93%	95.91%	-	96.43%	96.42%	96.57%
Realm Java	93.07%	93.33%	93.39%	93.26%	93.48%	93.56%
RxJava	66.30%	66.49%	66.33%	60.25%	61.45%	60.73%
Spring Boot	88.03%	88.06%	90.22%	87.71%	87.76%	89.91%
Spring Framework	99.59%	99.60%	99.61%	99.60%	99.55%	99.62%
Spring Security	97.89%	97.92%	97.91%	97.77%	97.79%	97.79%
Tomcat	97.49%	97.54%	97.53%	97.49%	97.53%	97.52%
Tutorials	91.50%	-	91.89%	92.14%	-	92.49%

A.6 Results of the TCC modeling

This section shows the results obtained for TCC modeling in Table A.6.

Table A.5: $\overline{R}^2$ values computed from the LCOM models.

System	lin.	quad.	cub.	log. 1	log. 2	log. 3
Alluxio	91.83%	92.50%	-	91.94%	92.72%	-
Antlr4	92.29%	92.68%	92.56%	92.32%	92.78%	92.77%
Arduino	87.29%	86.93%	86.07%	84.91%	84.05%	82.70%
Bazel	97.67%	97.67%	-	97.65%	97.65%	-
Bisq	93.29%	93.76%	93.87%	93.28%	93.81%	93.89%
Buck	95.21%	95.18%	95.30%	95.12%	95.09%	95.21%
CAS	93.89%	93.94%	94.05%	94.33%	94.38%	94.48%
CoreNLP	92.78%	92.75%	92.78%	92.95%	92.93%	92.95%
Dbeaver	98.66%	95.05%	98.77%	98.57%	94.82%	98.68%
Dropwizard	98.09%	98.13%	98.18%	98.17%	98.21%	98.25%
Druid	99.25%	99.26%	99.26%	99.32%	99.32%	99.33%
Eclipse JDT Core	98.73%	98.72%	98.72%	98.91%	98.90%	98.90%
Eclipse PDE UI	97.75%	52.14%	97.56%	96.60%	74.75%	73.13%
Elasticsearch	83.00%	81.22%	83.67%	77.88%	76.55%	78.27%
Equinox Framework	87.85%	87.98%	87.27%	87.32%	87.42%	86.63%
FrameworkBenchmarks	94.05%	93.01%	93.98%	94.06%	92.98%	93.99%
Gocd	34.63%	30.03%	42.90%	42.02%	42.79%	49.39%
Graylog	97.74%	97.83%	97.97%	98.15%	98.24%	98.35%
Guava	96.44%	96.44%	95.29%	96.47%	96.46%	95.27%
Hibernate Orm	93.73%	93.70%	93.79%	93.49%	93.45%	93.54%
J2ObjC	88.30%	88.56%	88.76%	88.04%	88.30%	88.50%
Jabref	96.93%	96.93%	96.97%	96.86%	96.87%	96.91%
Jenkins	95.23%	95.24%	95.30%	95.36%	95.37%	95.42%
Jitsi	82.77%	80.91%	86.21%	82.80%	80.94%	86.27%
JMeter	83.29%	84.98%	89.66%	83.45%	85.66%	89.79%
JUnit 5	98.87%	98.87%	98.90%	98.63%	98.63%	98.67%
K-9 Mail	97.81%	97.13%	97.88%	98.28%	97.46%	98.35%
Kafka	-	-	61.31%	-	-	61.27%
LanguageTool	96.72%	97.00%	96.89%	96.80%	97.13%	-
Lucene	58.81%	-	59.66%	29.96%	-	37.69%
MinecraftForge	57.91%	55.80%	55.56%	39.92%	39.06%	38.87%
Neo4j	98.89%	98.89%	98.88%	98.97%	98.97%	98.97%
Netty	67.85%	68.41%	69.47%	66.98%	67.50%	68.52%
OpenRefine	46.84%	46.85%	47.70%	36.71%	36.24%	37.38%
OrientDB	96.63%	96.65%	-	96.55%	96.57%	-
Pentaho Kettle	99.07%	99.09%	99.09%	99.08%	99.10%	99.10%
Pentaho Platform	95.13%	95.22%	95.28%	94.72%	94.82%	94.87%
Pinpoint	96.45%	96.46%	96.45%	96.97%	96.97%	96.96%
PMD	94.07%	92.93%	94.14%	94.94%	-	94.99%
Realm Java	96.31%	94.97%	96.29%	96.28%	94.97%	96.26%
RxJava	86.14%	86.18%	86.90%	83.37%	83.46%	84.62%
Spring Boot	78.69%	80.28%	80.03%	79.01%	80.68%	80.40%
Spring Framework	96.44%	98.25%	98.24%	96.38%	98.19%	98.18%
Spring Security	92.14%	92.18%	92.17%	91.91%	91.94%	91.93%
Tomcat	98.80%	98.79%	98.19%	98.85%	98.84%	98.25%
Tutorials	95.72%	95.80%	96.07%	95.47%	95.59%	95.90%

A.7 Results of the NOA modeling

This section shows the results obtained for NOA modeling in Table A.7.

Table A.6: $\overline{R}^2$ values computed from the TCC models.

System	lin.	quad.	cub.	log. 1	log. 2	log. 3
Alluxio	88.64%	86.58%	88.95%	89.79%	86.55%	89.84%
Antlr4	89.27%	89.26%	89.05%	89.25%	89.24%	89.04%
Arduino	92.71%	92.89%	92.82%	89.15%	85.92%	88.84%
Bazel	94.67%	93.05%	95.05%	94.59%	93.00%	94.98%
Bisq	91.23%	91.25%	91.65%	90.27%	90.32%	90.85%
Buck	95.28%	95.35%	95.56%	95.35%	95.43%	95.64%
CAS	98.37%	98.38%	98.38%	98.21%	98.23%	97.78%
CoreNLP	95.74%	95.75%	96.04%	95.66%	95.66%	95.72%
Dbeaver	88.82%	89.72%	89.99%	88.70%	89.60%	89.90%
Dropwizard	88.56%	89.71%	89.64%	88.21%	89.23%	89.17%
Druid	98.03%	98.02%	98.03%	98.01%	98.00%	98.01%
Eclipse JDT Core	98.31%	97.42%	98.34%	98.22%	97.32%	98.26%
Eclipse PDE UI	68.70%	68.65%	68.90%	64.58%	64.53%	64.78%
Elasticsearch	76.67%	-	77.21%	71.77%	61.74%	66.31%
Equinox Framework	95.69%	95.58%	95.75%	95.06%	94.59%	95.12%
FrameworkBenchmarks	-	96.04%	96.26%	-	96.08%	96.30%
Gocd	46.56%	44.38%	48.76%	49.22%	48.11%	54.18%
Graylog	93.14%	92.66%	93.16%	92.89%	89.38%	92.90%
Guava	96.88%	96.89%	96.94%	96.79%	96.80%	96.85%
Hibernate Orm	96.54%	96.23%	96.56%	96.44%	96.13%	96.46%
J2ObjC	98.23%	98.29%	98.32%	98.36%	98.41%	98.44%
Jabref	99.08%	99.09%	99.14%	99.03%	99.04%	99.08%
Jenkins	98.75%	98.75%	98.75%	98.74%	98.74%	98.74%
Jitsi	-	86.05%	-	-	86.24%	-
JMeter	88.39%	88.61%	87.66%	88.84%	89.03%	87.91%
JUnit 5	94.48%	97.45%	-	94.38%	97.32%	-
K-9 Mail	91.11%	94.58%	91.28%	90.56%	94.37%	90.78%
Kafka	95.08%	95.44%	-	95.14%	95.51%	-
LanguageTool	92.93%	93.05%	-	93.56%	93.67%	-
Lucene	60.41%	61.84%	-	44.11%	46.03%	48.76%
MinecraftForge	41.53%	43.91%	39.80%	33.70%	36.55%	34.63%
Neo4j	91.15%	89.69%	91.12%	91.25%	89.83%	91.22%
Netty	79.36%	80.48%	79.96%	78.22%	79.35%	78.79%
OpenRefine	-	-	63.19%	-	-	56.02%
OrientDB	97.96%	97.96%	97.99%	98.03%	98.03%	98.05%
Pentaho Kettle	99.77%	99.78%	99.78%	99.75%	99.77%	99.77%
Pentaho Platform	91.14%	-	89.79%	90.72%	-	89.29%
Pinpoint	98.10%	98.09%	98.16%	98.02%	97.99%	98.09%
PMD	95.39%	95.47%	-	95.76%	95.80%	-
Realm Java	86.40%	86.79%	86.64%	86.61%	86.92%	86.84%
RxJava	87.26%	86.88%	86.12%	84.83%	84.46%	82.98%
Spring Boot	95.67%	96.24%	-	80.27%	96.20%	-
Spring Framework	99.27%	99.26%	99.27%	99.26%	99.26%	99.27%
Spring Security	98.86%	98.86%	98.87%	98.84%	98.84%	98.85%
Tomcat	93.53%	93.38%	93.37%	93.43%	93.27%	93.26%
Tutorials	99.30%	99.38%	99.37%	98.93%	99.04%	99.07%

A.8 Results of the NOM modeling

This section shows the results obtained for NOM modeling in Table A.8.

Table A.7: $\overline{R}^2$ values computed from the NOA models.

System	lin.	quad.	cub.	log. 1	log. 2	log. 3
Alluxio	91.38%	92.34%	-	91.54%	92.66%	-
Antlr4	97.89%	97.89%	97.93%	98.46%	98.46%	98.49%
Arduino	97.73%	97.68%	97.89%	98.45%	98.35%	98.37%
Bazel	95.25%	95.22%	93.80%	95.12%	95.09%	93.60%
Bisq	92.99%	93.62%	-	92.40%	93.16%	-
Buck	90.85%	90.78%	90.71%	91.12%	91.05%	90.99%
CAS	98.77%	97.98%	98.81%	98.96%	98.97%	98.98%
CoreNLP	96.01%	96.09%	96.16%	96.09%	96.15%	96.23%
Dbeaver	98.05%	96.63%	98.13%	97.94%	96.43%	98.02%
Dropwizard	97.68%	97.91%	97.91%	97.89%	98.10%	98.11%
Druid	98.87%	98.97%	99.01%	99.00%	99.03%	99.07%
Eclipse JDT Core	96.95%	96.95%	96.94%	97.32%	97.31%	97.31%
Eclipse PDE UI	87.82%	87.88%	87.04%	86.67%	86.57%	86.13%
Elasticsearch	77.75%	77.69%	80.74%	73.52%	73.42%	76.92%
Equinox Framework	91.05%	91.15%	90.51%	90.53%	90.63%	89.92%
FrameworkBenchmarks	92.25%	89.58%	-	92.32%	89.65%	-
Gocd	44.17%	42.18%	-	51.22%	52.86%	55.73%
Graylog	97.48%	97.61%	97.71%	97.12%	97.65%	97.78%
Guava	99.33%	99.40%	99.75%	99.25%	99.30%	99.25%
Hibernate Orm	89.68%	89.85%	89.94%	90.15%	90.28%	90.40%
J2ObjC	86.88%	87.12%	87.26%	87.57%	87.78%	87.92%
Jabref	99.45%	99.47%	99.31%	99.46%	99.48%	99.48%
Jenkins	95.19%	95.08%	95.17%	95.30%	95.20%	95.29%
Jitsi	91.67%	92.13%	94.47%	91.79%	92.21%	94.53%
JMeter	78.50%	77.54%	78.75%	78.88%	77.84%	78.99%
JUnit 5	95.26%	95.28%	93.75%	95.57%	95.58%	94.21%
K-9 Mail	95.07%	93.75%	95.19%	94.99%	94.62%	95.64%
Kafka	92.64%	-	92.41%	92.88%	-	92.67%
LanguageTool	92.94%	93.04%	92.90%	93.01%	93.08%	92.94%
Lucene	4.58%	9.79%	19.65%	3.31%	6.99%	10.67%
MinecraftForge	60.13%	60.01%	-	37.22%	34.42%	32.66%
Neo4j	98.00%	98.00%	97.99%	97.98%	97.97%	97.96%
Netty	76.23%	77.04%	77.80%	74.82%	75.62%	76.36%
OpenRefine	39.52%	-	40.16%	32.82%	-	32.01%
OrientDB	99.14%	99.14%	99.15%	99.14%	99.14%	99.15%
Pentaho Kettle	97.58%	97.73%	97.47%	97.52%	97.67%	97.40%
Pentaho Platform	86.81%	-	85.35%	86.52%	-	84.97%
Pinpoint	98.44%	98.81%	98.79%	98.44%	98.90%	98.85%
PMD	87.57%	88.55%	89.52%	87.46%	88.50%	89.46%
Realm Java	96.17%	96.16%	96.16%	96.07%	96.06%	96.07%
RxJava	82.62%	82.96%	82.64%	80.24%	81.02%	80.90%
Spring Boot	80.38%	82.64%	83.32%	80.41%	82.82%	83.50%
Spring Framework	97.07%	97.12%	97.11%	96.98%	97.03%	97.02%
Spring Security	99.43%	99.43%	99.45%	99.42%	99.42%	99.43%
Tomcat	99.86%	99.86%	99.87%	99.87%	99.87%	99.87%
Tutorials	98.47%	98.56%	98.67%	98.29%	98.38%	98.58%

Table A.8: $\overline{R}^2$ values computed from the NOM models.

System	lin.	quad.	cub.	log. 1	log. 2	log. 3
Alluxio	81.63%	83.25%	-	82.00%	83.53%	-
Antlr4	96.72%	96.96%	97.00%	96.70%	97.21%	97.00%
Arduino	95.14%	95.38%	94.71%	95.81%	96.02%	95.53%
Bazel	97.70%	96.85%	97.72%	97.66%	97.67%	97.68%
Bisq	87.22%	90.38%	85.41%	87.14%	90.23%	85.20%
Buck	98.08%	96.90%	98.10%	97.86%	96.65%	97.87%
CAS	92.93%	93.78%	94.00%	93.10%	94.01%	94.22%
CoreNLP	96.42%	96.63%	96.66%	96.16%	96.42%	96.48%
Dbeaver	98.06%	98.12%	98.17%	98.02%	98.08%	98.13%
Dropwizard	93.95%	94.29%	94.53%	94.41%	94.72%	94.94%
Druid	99.57%	99.57%	99.58%	99.65%	99.65%	99.66%
Eclipse JDT Core	98.62%	98.61%	98.61%	98.91%	98.91%	98.90%
Eclipse PDE UI	87.07%	88.70%	86.37%	86.30%	87.97%	85.41%
Elasticsearch	49.14%	49.61%	40.28%	45.53%	46.00%	36.80%
Equinox Framework	98.42%	98.42%	98.45%	98.52%	98.26%	98.55%
FrameworkBenchmarks	94.92%	94.90%	-	94.96%	95.02%	-
Gocd	51.61%	52.81%	56.45%	52.63%	51.80%	-
Graylog	94.95%	94.98%	95.10%	94.45%	94.49%	94.61%
Guava	99.52%	99.54%	99.56%	99.51%	99.53%	99.54%
Hibernate Orm	97.97%	97.97%	98.05%	97.74%	97.73%	97.82%
J2ObjC	80.30%	80.60%	81.65%	81.96%	82.17%	83.19%
Jabref	98.26%	98.24%	98.22%	98.29%	98.31%	98.27%
Jenkins	96.80%	96.82%	96.88%	96.81%	96.84%	96.89%
Jitsi	92.96%	-	94.74%	92.98%	-	94.75%
JMeter	87.47%	88.25%	-	87.32%	88.14%	-
JUnit 5	98.40%	98.41%	98.47%	98.22%	98.23%	98.29%
K-9 Mail	-	80.18%	-	-	81.85%	-
Kafka	89.74%	-	93.14%	89.85%	-	93.26%
LanguageTool	94.63%	94.65%	95.35%	94.63%	94.65%	95.17%
Lucene	-	-	-	21.42%	-	-
MinecraftForge	43.16%	46.28%	-	32.68%	35.07%	34.81%
Neo4j	84.92%	84.87%	84.92%	85.01%	84.96%	85.01%
Netty	80.18%	80.08%	80.72%	80.24%	80.14%	80.72%
OpenRefine	-	-	34.98%	-	-	-
OrientDB	97.21%	97.22%	97.21%	97.35%	97.36%	97.36%
Pentaho Kettle	98.57%	98.59%	98.59%	98.62%	98.63%	98.63%
Pentaho Platform	94.12%	94.10%	94.11%	93.96%	93.94%	93.94%
Pinpoint	91.87%	91.85%	91.86%	91.46%	91.43%	91.45%
PMD	96.39%	96.37%	96.43%	97.30%	97.29%	97.34%
Realm Java	95.26%	95.31%	95.10%	95.53%	95.59%	95.53%
RxJava	74.31%	76.54%	74.83%	65.30%	68.57%	66.58%
Spring Boot	97.47%	88.67%	97.55%	97.49%	87.63%	97.58%
Spring Framework	96.37%	98.02%	98.04%	96.30%	97.95%	97.97%
Spring Security	99.53%	99.53%	99.53%	99.52%	99.52%	99.53%
Tomcat	99.68%	99.70%	99.68%	99.68%	99.69%	99.69%
Tutorials	98.02%	98.07%	98.37%	97.69%	97.74%	98.06%