# MODELAGEM E PREDIÇÃO DA EVOLUÇÃO DOS ATRIBUTOS INTERNOS DE QUALIDADE DE SOFTWARE

BRUNO LUAN DE SOUSA

# MODELAGEM E PREDIÇÃO DA EVOLUÇÃO DOS ATRIBUTOS INTERNOS DE QUALIDADE DE SOFTWARE

Projeto de tese apresentado ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Orientadora: Mariza Andrade da Silva Bigonha
Coorientadoras: Kecia Aline Marques Ferreira, Glaura da Conceição Franco

Belo Horizonte

Agosto de 2020

BRUNO LUAN DE SOUSA

# MODELING AND PREDICTING EVOLUTION OF SOFTWARE QUALITY INTERNAL ATTRIBUTES

Thesis project 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: Mariza Andrade da Silva Bigonha
Co-Advisors: Kecia Aline Marques Ferreira, Glaura da Conceição Franco

Belo Horizonte

August 2020

# Acknowledgments

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

First of all, I thank God for always being my guide, illuminating my paths, and never letting me give up even in the most difficult moments that I encountered along this journey.

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

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

I thank my advisors 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 these three years of the Ph.D. course.

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

Finally, I thank everyone who has contributed directly or indirectly for 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 atualizar os sistemas de software. Esta tese proposta se concentra em investigar como sistemas de software evoluem ao longo do tempo. Muitos estudos têm investigado este tópico, contudo, não existe até o momento uma visão geral e consolidada do estado da arte da pesquisa em evolução de software. Devido a esse fato, o primeiro passo desta pesquisa foi conduzir uma Revisão Sistemática da Literatura (RSL) abrangente a fim de compilar o conjunto de conhecimentos sobre a evolução de software e entender como esse tópico está sendo investigado. Esta RSL encontrou 130 artigos neste assunto, publicados de 1979 a 2019. A análise desses estudos revelou que a evolução de software tem sido estudada sob cinco perspectivas: (i) verificação da aplicabilidade das leis de Lehman; (ii) proposta de aplicações; (iii) análise da evolução com foco na qualidade; (iv) análise da evolução estrutural do software; e (v) proposta de modelos para evolução de software. Além disso, os estudos em Engenharia de Software têm confirmado que, à medida que os sistemas de software evoluem, eles se tornam cada vez mais complexos e difíceis de manter. Entretanto, os estudos realizados até o momento não detalham como ocorre essa degradação. Entender como a estrutura interna do software evolui é essencial para ajudar desenvolvedores a melhor planejar, gerenciar e executar tarefas de manutenção de software.

Este trabalho visa fornecer um conhecimento refinado de como a estrutura interna dos sistemas de software orientados por objetos evolui. Foram consideradas três características internas de sistemas de software orientado por objetos: acoplamento, tamanho das classes e hierarquia de herança. Foi definido um novo método baseado em análise de séries temporais, técnicas de regressão linear e testes de tendência para analisar a evolução de sistemas orientado por objetos. Aplicando essa abordagem, identificou-se as funções que melhor explicam como o acoplamento, o tamanho das classes e a árvore de herança evoluem. Para avaliar essas características, utilizaram-se métricas de software definidas na literatura e consideraram-se dados de 10 projetos de código aberto baseados em Java. Os dados compreendem um período de 2001 a 2011

e consideram de 72 a 248 *releases* de projetos. Os principais resultados deste trabalho até o momento são:

1. um novo método para analisar evolução de software baseado na análise de séries temporais;
2. a identificação das funções que explicam a evolução do acoplamento, do tamanho de classes e da hierarquia de herança;
3. a identificação de um conjunto composto por 15 propriedades relacionadas à evolução dessas características internas do software orientado por objetos.

Na sequência deste projeto de tese, objetiva-se definir e avaliar um método de predição para evolução de software orientado por objetos em termos de acoplamento, tamanho de classes e hierarquia de herança. O método será baseado nos resultado encontrados na primeira parte desta pesquisa. Será construído uma abordagem automática para identificar um modelo de predição para um dado sistema. A entrada deste método será um conjunto de dados evolucionários de métricas de software referentes a um sistema de software, ou seja, séries temporais de métricas. O resultado do método será um modelo que prevê como o sistema de software evolui em termos dos atributos que as métricas medem. Para avaliar o método proposta, o *dataset* utilizado na primeira parte deste projeto de tese será ampliado. Em cenários reais da Engenharia de Software, os resultados da tese proposta podem ajudar os desenvolvedores a planejar suas estratégias para acomodar mudanças e novos recursos no sistema, de modo que a degradação da arquitetura do 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 process of the software life cycle. It consists of adapting, maintaining, and updating software systems. This thesis proposal concentrated on investigating how software systems evolve along time. Many studies have been carried out on this topic. However, we do not have a general view of the state-of-the-art of software evolution research. Due to this fact, our first step was to carry out a comprehensive Systematic Literature Review (SLR) to compile the body of knowledge on software evolution and understand how the literature investigated this topic. Our SLR identified 130 papers in this subject, published from 1979 to 2019. The analysis of those studies revealed that software evolution has been studied from five perspectives: (i) verification of the applicability of Lehman's laws; (ii) proposal of applications, (iii) analysis of the evolution with a focus on quality, (iv) analysis of the software structure evolution, and (v) proposal of models for software evolution. Besides, the studies on software engineering have confirmed that as software systems evolve, it becomes increasingly complex and challenging to maintain. Nevertheless, the studies carried out so far have not detailed how such degradation occurs. Understanding how the internal software structure evolves is essential to help developers to better plan, manage, and perform software maintenance tasks.

This work aims to provide a fine-grained knowledge of how the internal structure of object-oriented software systems evolves. We consider three internal characteristics of object-oriented software systems: coupling, size of classes, and inheritance hierarchy. We defined a novel method based on time series analysis, linear regression techniques, and trend tests to analyze the evolution of object-oriented systems. Applying such an approach, we identified the function that better explains how the coupling, classes' size, and the inheritance tree evolve. To assess these characteristics, we used software metrics defined in the literature and considered data from ten Java-based open-source projects. The data comprises a period from 2001 to 2011 and consider s 72 to 248 releases of the projects. The main results of this work until now are the following:

1. a novel method to analyze software evolution based on time series analysis;
2. the creation of a function that explains the evolution of coupling, size of classes, and inheritance hierarchy;
3. a set of 15 properties regarding the evolution of these internal characteristics of object-oriented software.

In the sequel of this thesis project, we aim to define and evaluate a prediction method for object-oriented software evolution in terms of coupling, size of classes, and inheritance hierarchy. We will base the method on the results found in the first part of this research. We will construct an automatic approach to identify a prediction model for a given system. Our method's entry is a set of evolutionary data of software metrics of a given software system, i.e., the metric time series. The result of the method will be a model that predicts how the software system will evolve in terms of the attributes the metrics measure. To evaluate the proposed method, we will extend the data set we used in the first part of this thesis project. In real software engineering scenarios, the results of the proposed thesis may support developers to plan their strategies to accommodate changes and novel features in the system, so that the software architecture degradation may be mitigated or avoided.

**Palavras-chave:** Software Evolution, Software Metrics, Software Quality, Time Series, Trend Analysis.

# List of Figures

# List of Tables

# Contents

# Chapter 1

# Introduction

Software evolution is a general term employed in Software Engineering for describing one of the phases of the software 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 in the system, an inclusion or a change in the system requirements. Mens et al. [2010] consider software evolution and maintenance synonyms since they are related over the software life cycle and deal with similar activities, such as updating and fixing the system after its first release. Software evolution and the changes caused by them are essential because they allow including and enhancing novel features in a system to meet the demands required by its users or clients. However, the continuous changes generate, in most cases, an increase of the complexity of the system's internal structure, which may lead to high costs to accommodate changes and the 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]. Then, the way how dealing with changes in the software structure during this phase is a practice that requires much attention from developers and software engineers. Any change needs to be carefully planned and studied before being implemented so that there is no degradation of the system's architecture and, consequently, further increase software costs.

The way software systems evolve has been a subject of research in Software Engineering for decades. As a starting point, Lehman et al. [1997] carried out a set of empirical studies aiming to better understand the software evolution characteristics. They describe the evolutionary nature of the software and conclude that, in general, software grow and undergo maintenance continuously, have increasing complexity and

decreasing quality over its evolution. They summarize their findings as eight laws, which are widely known as Lehman's laws. Table 1.1 presents the eight software evolution laws proposed by Lehman et al. [1997].

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 progrssively 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 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 an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behaviour 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 through by rigorous maintenance and adaptation to operational environment changes. |
| VIII | Feedback System | The evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base. |

Lehman's laws are one of the landmarks on software evolution, and they have inspired other works to investigate this topic in the last years. A large part of the studies existing in the literature aim to check and validate the presence of the Lehman's laws in the software development contexts, such as open-source systems [Lee et al., 2007b; Mens et al., 2008; Xie et al., 2009; Businge et al., 2010; Israeli and Feitelson, 2010; Alenezi and Almustafa, 2015], mobile applications [Zhang et al., 2013; Li et al., 2017; Gezici et al., 2019], proprietary software [Barry et al., 2007] and C library [Gonzalez-Barahona et al., 2014]. Besides, other works have aimed to characterize software evolution under

some software dimensions, especially regarding software size [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., 2007; Koch, 2007; Gonzalez-Barahona et al., 2009; Hatton et al., 2017].

The studies on the evolution of size have diverged in their results and, consequently, it has not yet a clear and precise conclusion about the evolution of this dimension in software systems. Therefore, there is still a gap in the comprehension on how the internal software dimensions, such as size, coupling, inheritance hierarchy, among others, evolve. For instance, it is widely known that the internal quality declines, and the complexity increases in a software system when it evolves. However, so far, there is not a pattern explaining how software systems' internal structure degrades in a fine-grained view about.

This thesis project aims to provide a detailed view of the evolution of some internal dimensions in software systems. Besides, this work has as goal using this knowledge to build prediction models for software evolution. For this purpose, we divided this thesis project into three parts. In the first one, we carried out a Systematic Literature Review (SLR) aiming to compile the works on software evolution done so far. In this first part, we (i) provided an overview of the state-of-the-art regarding software evolution, (ii) found the main research lines about this topic, (iii) characterized the main software evolution datasets, and (iv) identified the main gaps existing in the literature.

In the second part of this thesis project, we aim to study and characterize object-oriented software systems' evolution from the perspective of some dimensions, such as coupling, size, and inheritance hierarchy. Coupling is a dimension that describes the level of dependence between the modules of a software system [Myers, 1975]. 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]. 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].

To characterize these dimensions, we used time series with data of six software metrics. We considered fan-in and fan-out to represent coupling, 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 indicates the number of references made to a given class by other classes, while fan-out reflects the number of calls made by a given class to other classes [Sommerville, 2012]. NOA and NOM are the numbers of attributes and methods of a class [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 define a novel method based on time series analysis, linear regression techniques, and trend tests to analyze object-oriented systems' evolution. We applied this method in a dataset, composed of evolution data regarding ten object-oriented software systems, and identified 15 properties that describe the behavior of the analyzed dimensions over software evolution.

In the third part of this thesis project, we intend to use the software evolution properties identified in the first part as a background in the definition of a prediction method for object-oriented software evolution. Our method will build models that predict and project the evolution of a specific system in terms of coupling, size of classes, and inheritance hierarchy. To propose this method, we will define an automatic approach to identify a prediction model for a given system. Then, our method's input will be a set of evolutionary data regarding software metrics from a given software system, i.e., the metric time series. The result of the method will be a model that predicts how the software system will evolve in terms of the attributes the metrics measure. Our objective with these models is to provide developers a way to monitor and track their software evolution. They will then be able to plan their strategies better to accommodate changes and novel features in the software and avoid software architecture degrading over their evolution.

We will extend the software evolution dataset used in the first part of this thesis project. The resulting dataset will be applied to evaluate our prediction method we will design in the next steps of this research. The dataset is called `COMETS` and comprises data of software metrics regarding the evolution of ten object-oriented software. We considered this dataset in this research because it is the largest dataset in the literature regarding software metrics and the number of systems. However, the most recent information that it stores is from December $11^{st}$, 2011. Therefore, we intend to update it by adding values of metrics regarding recent releases from their software systems and use these data to support us to assess our prediction models.

This first and second parts of this Ph. D. research are concluded. The results are reported throughout this document. The *blue*third part of this research will be carried out over the next year of the Ph. D. course.

## 1.1 Goal

The goals of this thesis research are:

1. Carry out a Systematic Literature Review to compile the corpus of knowledge on software evolution.

2. Define an analysis method to study the evolution of internal dimensions regarding the software structure.

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

4. Extend a software evolution dataset containing software metrics' time series.

5. Build prediction models of the following software dimensions: size, coupling, and inheritance.

## 1.2 Contributions

This research provides the following contributions so far:

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

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

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

In the final Ph.D. thesis, we aim to achieve additional contributions:

1. An extended dataset with recent data regarding the evolution of the object-oriented software systems.

2. A prediction method that extracts forecast models from the evolutionary data regarding software metrics for object-oriented software systems.

## 1.3 Publications

The research reported in this thesis project has generated the following publications and papers submitted that are under review:

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). Although not presented in this document, the work on Software Engineering evolution provided an overview to the author of this work on how software evolution research has increased.

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 Software Components, Architectures, and Reuse (SBCARS '19), Salvador, BA, Brazil, pages 23–32, 2019. (Published and awarded as the 2*nd* best paper from the symposium)

3. Sousa, B.L.; Bigonha, M.A.S; Ferreira, K.A.M.; Franco G.C. Evolution of Size and Inheritance in Object-Oriented Software – A Time Series Based Approach. Submitted to an international journal, pages 1–11, 2020. (Under Review)

4. Sousa, B.L.; Bigonha, M.A.S; Ferreira, K.A.M.; Franco G.C. A Comprehensive Systematic Literature Review of Software Evolution. Submitted to an international journal, pages 1–35, 2020. (Under Review)

## 1.4 Organization of the Thesis Project

The remainder of this thesis project is organized 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 existing in the literature.

Chapter 3 describes the systematic literature review (SLR) on software evolution we carried out to compile the knowledge existing in the literature about this topic.

Chapter 4 presents and describes the novel method for analysis of software evolution based on time series.

Chapter 5 reports the main observations and results we found about the evolution of coupling, size, and inheritance hierarchy in object-oriented software systems.

Chapter 6 summarizes the evolution properties we extracted, considering the observation exhibited in Chapter 5.

Chapter 7 shows the next steps to conclude this thesis research and gives directions to achieve the objectives proposed for the second part of this Ph. D. work.

Chapter 8 concludes this thesis project and presents some proposals for future works.

# Chapter 2

# Background

This chapter presents the main concepts applied in this thesis project. They are object-oriented software metrics (Section 2.1) and time series (Section 2.2).

## 2.1 Object-Oriented Software Metrics

In the context of software internal structure, metrics are used to evaluate software dimensions such as modularity, complexity, size, coupling, and cohesion. Our recent study analyzed a large number of articles published in the International Conference on Software Engineering (ICSE), a relevant Software Engineering conference. The conclusion is that software metrics are the study's object comprising the ten most explored topics of the area [Sousa et al., 2019]. Since the 1990s, the literature has provided many novel units of measure 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 the sets proposed by Chidamber and Kemerer [1994]. They are widely known as CK, representing the landmarks of this field of knowledge.

There are several object-oriented software metrics proposed in the literature. As this study is based on object-oriented software metrics, we surveyed the main ones described in the literature. In this section, we present the CK and and the MOOD metrics in Sections 2.1.1 and 2.1.2, respectively. In Sections 2.1.3 and 2.1.4 we present the Martin's metrics [Martin, 1994] and some complexity indicators. Sections 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 metrics 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 a group 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 is necessary to assign weights for each class' method, which may be inferred by a secondary metric. Chidamber and Kemerer [1994] do not define a specific complexity indicator for using together with this metric, and then, the weights may be defined both via lines of code and cyclomatic complexity of each method. According to the authors, WMC indicates efforts of development and maintenance of 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 harms its cohesion aspect.

- **DIT (Depth of Inheritance)** is related to the inheritance hierarchy in object-oriented software. It represents the level in 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 are indicators of 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 that 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 in terms of tests to avoid that errors introduced into the superclass propagate over its subclasses.

- **LCOM (Lack of Cohesion Of Methods)** is related to cohesion in object-oriented software. It uses the notion of degree of similarity 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. They - then 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 none 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 the cohesion degree of this respective class.

- **CBO (Coupling Between Object)** measures the number of classes that it calls through an association relationship. The association between two classes may occur when one of them 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 level of coupling have a low potential of reuse and are more prone to change when other parts of the system are modified.

- **RFC (Response For 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

The MOOD metrics were proposed by Abreu and Carapuça [1994] to measure the following aspects of the object-oriented software: inheritance hierarchy, encapsulation, coupling, polymorphism, and software reuse. An essential aspect 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 metrics composes 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 it 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 in the system. As well as to MIF, AIF values close to 1 indicate a high degree of reuse of data.

- **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 there is a class responsible for providing services and the other type 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 that the system has a low consumption of this resource.

- **MHF (Method Hiding Factor)** indicates the percentage of methods hidden in the system. It is a metric computed to the system level that consists of dividing 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. The closer to 0 the value of this metric, the more the number of public methods to the system's users. Such a scenario is an indication of a low degree of encapsulation of the system.

- **AHF (Attribute Hiding Factor)** is similar to MHF, but it considers hidden attributes rather than hidden methods. We compute this metric by the ratio between the number of hidden attributes in all classes of the system 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, for software quality, the ideal scenario is that systems have values equal or as close to 0 as possible.

### 2.1.3 Martin Metrics

This section brings the metrics proposed by Martin [1994]. To propose these metrics, Martin [1994] defined a concept that he labeled of class category. A class category is a cohesive group of classes in which: if one class existing in this category is changed, the other classes also have a high chance of being changed; classes are reused together; classes have a common goal or perform interdependent functions. This group is composed of five metrics, and we describe them as follows.

- **AC (Afferent Coupling)** indicates the number of external classes that consume the classes 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 to a given category that depends on external classes to this category. According to Martin [1994], this metric is another indicator of coupling. 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 the ratio between the number of abstract classes within a category and the total number of classes belonging to this category.

- **RMD (Normalized Distance)** measures the distance of instability (I) and abstraction (A) by using the 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 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 degree of balance between instability and abstraction.

### 2.1.4 Complexity Metrics

This section presents some 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 values of fan-in mean that 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], 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, such as conditional (`if`) and repetition loops (`for` and `while`), are inserted one inside each other. NBD is indicative of 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 a 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 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.

### 2.1.5 Lorenz and Kidd's Metrics

This section brings 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 composes it, and all of them, except PAR, is 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 may characterize the size of a class and the number of services it provides.

- **NAP (Number of Public Attributes)** measures the number of public attributes existing in a class. 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 protect 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 protect 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 as static in the classes.
- **PAR (Number of Parameters)** measures the total number of parameters regarding each method belonging to 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 aiming to fill the gaps left open by the CK metrics. The metrics defined by Li [1999] evaluates 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 provide support to multiple inheritances, e.g., Java, DIT provides efficient measures. However, the same does not occur in software built-in languages that provide 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 that aims to measure the number of classes that precede a particular class in an inheritance hierarchy, and 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 little 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 a 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 the class A. The NDC metric also measures the number of descendants of a given class. However, it analyzes all posterior levels of the analyzed 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 for others, e.g., public methods. According to Li [1999], NLM refers to the size aspect. However, 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 they differ in the way 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 the development and maintenance costs of a class than the 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 a 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 that were 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 that it consumes from

another. With this metric, class A has a service coupling with a class B when A invokes one or more methods defined in B. Therefore, CTM indicates the total of messages that a particular class shares with other types through the method call mechanism.

### 2.1.7 Malik & Chhillar 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 by 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 a 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 more complex and prone to fail.

- **ACI (Average Complexity by Inheritance)** consists of an arithmetic average of the CCI values regarding all classes of the software. 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 some 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 provides a comparison between these two categories of metrics.

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

| Static Metrics | Dynamic Metrics |
|---|---|
| Simpler to collect | Difficult to obtain |
| Available at the early stages of software development | Accessible very late in 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 to deal with dead code and OO features such as inheritance, polymorphism and dynamic binding | Dynamic metrics are capable to deal with all object-oriented features and dead code |
| Less precise than dynamic metrics for the real-life systems | More precise than static metrics for the real-life systems |

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

#### 2.1.9.1 Coupling

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

- **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 the 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 the 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.

  To facilitate understanding the concept of these metrics, Arisholm et al. [2004] defined their nomenclature considering the orthogonal dimensions. 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* indicates 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 next 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. 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, we may infer that 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 aim 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 important 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. To extract this metric, it is necessary to observe the history of the object, i.e., the sequence of its states in time. Then, during a period

provided before starting to collect this metric, DCM is extracted by the sum over all program execution steps and the sum over the total number of objects coupled to the analyzed object.

#### 2.1.9.2 Cohesion

We describe here the dynamic metrics suites proposed to measure the cohesion dimension.

- **Gupta's Metrics** involve two dynamic cohesion metrics: Strong Functional Cohesion (SFC) and Weak Functional Cohesion (WFC). Gupta and Rao [2001] based on the concept of 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 outputs variables. Similarly, the WFC consists of the measure arising out of def-use pairs of each type found in dynamic slices 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 by the same way that the static LCOM, but 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 that it is accessed at runtime.

#### 2.1.9.3 Complexity

We describe here the dynamic metrics suites proposed to measure complexity.

- **Munson's Metrics** measure 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, as well as 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 occurs 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 observations made sequentially over time [Morettin and Toloi, 2006; Bowerman and O'Connell, 1993]. In general, the observations in a time series are serially correlated, and the main concern when working with this kind of data is to identify 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 the 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 observations made in fixed time intervals, i.e., observation intervals that belongs to a discrete set, e.g., the number of failures per software release. In contrast, a continuous

time series consists of observations continuously made over a specific time, e.g., daily temperature values.

In Software Engineering, there are works in which time series is applied as sequence of values regarding software metrics with the purpose to support the defect prediction [Couto et al., 2014] and analysis of the internal structures of software over the evolution process [Herraiz et al., 2006; Koch, 2007; Israeli and Feitelson, 2010].

During the creation of a time series, the observation intervals must be equally spaced out over the total time period in both continuous and discrete series. In cases where the time series interval is not equally spaced out, the final analysis of this data may lead to erroneous conclusions if specific forms of modeling able to deal with this problem are not chosen.

## 2.3 Final Remarks

There are many software metrics proposed in the literature, especially metrics for measuring object-oriented software. Among the several suites available, the ones most known and used are the CK Chidamber and Kemerer [1994]. In this study, we focused on investigating coupling, cohesion, size, and inheritance evolution. Due to this, we applied the following software metrics: fan-in and fan-out for coupling, DIT and NOC for inheritance hierarchy, and NOA and NOM for size.

Time series consists of a collection of observations made sequentially over time. In this work, we apply time series of software metrics to track how a particular software aspect evolves.

Chapter 3 presents a Systematic Literature Review (SLR) that aims to provide an overview of the state-of-the-art on software evolution. It also details the main lines of research on this topic by showing its main findings from the studies and gaps that have not yet been covered.

# Chapter 3

# Systematic Literature Review

This chapter presents a systematic literature review (SLR) aiming to identify and analyze relevant primary studies on software evolution, and compile the content produced on this topic in a broader view. We considered in this SLR studies that focus on the evolution of the source code and structure of software systems to: (i) provide an overview about the state-of-the-art regarding software evolution; (ii) identify the main research lines existing within this topic; (iii) extract and characterize the main types of the datasets on software evolution provided in the literature; and (iv) identify the main findings and gaps in the literature.

According to Kitchenham and Charters [2007], a systematic literature review (SLR) consists of a vehicle of identifying, evaluating, and interpreting all the relevant evidence of a specific topic, issue, or phenomenon of interest. Besides, the goals and motivations of an SLR presented by Kitchenham and Charters [2007] are:

- summarizing the technology or studied area to understand its limitations and benefits
- identifying gaps that were not yet covered by studies in a particular area
- providing new structures, directing new areas of research
- studying technologies and theories to validate theses or raise new hypotheses for studies.

A SLR differs from the non-systematic process because it is carried out formally and rigorously [Biolchini et al., 2005]. This formalism and rigor of the SLR force the process of conduction of this study to follow a well-defined protocol and establish a precise sequence of steps.

According to Biolchini et al. [2005], we must define the systematic literature review by a central research question that reflects its primary goal, and express it via terms and concepts regarding the proposed question. One of the main advantages of a SLR is to support other researchers to reproduce the defined methodology and, consequently, judge the steps and decisions taken in the research.

The studies that contribute to the conduction of the SLR and answer the defined research questions are known as primary studies [Kitchenham and Charters, 2007]. A secondary study is the one that reviews and analyzes the primary studies to identify and establish conclusions via results that are common between them. Due to this, the systematic literature review is considered a secondary study.

A systematic literature review protocol may be summarized into three steps: planning, execution, and analysis of the results. The planning consists of establishing the SLR goal and defining protocol with the steps that will be followed during the execution of the process. The execution consists of applying the protocol described in the planning phase. This phase is responsible for seeking the primary studies, filtering them using inclusion and exclusion criteria, and extracting the relevant information to answer the research questions. Finally, the analysis of the results is the phase where the data obtained from the primary studies will be analyzed and summarized to answer the research questions and publish the results.

Biolchini et al. [2005] proposed a model composed of five steps to guide researchers in conducting systematic literature reviews as follows.

1. **Definition of research questions.** It consists of clearly defining the research question of the SLR, considering the goal of the research.

2. **Selection of sources.** It aims to determine the search repositories where the primary studies will be sought. Besides, it specifies other decisions regarding the primary studies' language and definition of the search string for finding papers in electronic databases.

3. **Selection of the studies.** It consists of defining the inclusion and exclusion criteria and how the studies will be selected.

4. **Data extraction.** It aims to apply the inclusion and exclusion criteria in the primary studies and detail the selection process.

5. **Summarization of the results.** It consists of analyzing the relevant information extracted from the primary studies and presenting their main findings.

We organize the remainder of this chapter as follows. Section 3.1 presents the planning phase of the SLR. Section 3.2 describes the execution phase by presenting the steps and results obtained during the selection process of primary studies. Section 3.3 presents the main findings of this SLR by answering the proposed research questions. Section 3.4 shows the main threats to the validity of this study and discusses the main decision we have taken to mitigate them. Section 3.5 concludes this chapter by highlighting the main findings and contributions provided by this SLR.

## 3.1 Planning

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

### 3.1.1 Research Questions

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

Initially, we defined two general-purpose research questions.

**RQ1:** How has the literature approached studies on software evolution?

**RQ2:** What are the main features of the datasets used in studies on software evolution?

During the execution of this SLR, we realized that researchers had approached software evolution in different ways. Therefore, we classified the studies based on their focus and identified five research lines that we named as *categories* in this study. They are (i) applications; (ii) applicability of Lehman's laws; (iii) evolution of quality attributes; (iv) evolution of software structure; and (v) model. To detail these categories, we defined some specific research questions for each one of them. Table 3.1 summarizes the specific research questions of the present study.

### 3.1.2 Electronic Databases

Table 3.2 lists the electronic databases we used in this work. We chose them because they are virtual libraries with an extensive collection of full works and metadata from published researches at conferences and journals of great importance to the academic community.

Table 3.1: Specific Research Questions.

| Categories | Research Questions (RQ) |
|---|---|
| Applicability of Lehman's Laws | **RQA.1:** Which Lehman's Laws have been validated? |
| | **RQA.2:** In which application contexts the Lehman's laws have been analyzed? |
| Application | **RQB.1:** What are the software evolution applications proposed or used in literature? |
| | **RQB.2:** What are the main features of these applications? |
| Evolution of Quality Attributes | **RQC.1:** Which quality attributes have been analyzed in studies on software evolution? |
| | **RQC.2:** Which software metrics have been considered to analyze the evolution of software quality attributes? |
| Software Structural Evolution | **RQD.1:** Which dimensions of software structure have been evaluated in the literature? |
| | **RQD.2:** What are the main insights reported in the literature regarding software structure evolution? |
| Models | **RQE.1:** What are the types of models on software evolution proposed in the literature? |
| | **RQE.2:** Which techniques have been used in the models of software evolution? |
| | **RQE.3:** Which metrics have been defined in the literature to assess the accuracy of software evolution models? |
| | **RQE.4:** Which software aspects have been considered in software evolution models? |

Table 3.2: 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/` |

### 3.1.3 Search String

To identify relevant papers about the evolution of software structure and code, we formulated a search string with terms related to the topic stated in this SLR. Initially, we defined the keywords "software evolution", "software structure", and "source code" as the main terms of our expression. After that, we searched for synonyms of these terms to refine this expression and identify relevant and coherent studies able to answer the proposed research questions. The final search string is defined as follows.

```
("software evolution" OR "system evolution" OR "program evolution" OR
"structural evolution") AND ("architecture" OR "software structure"
OR "system structure" OR "program structure") AND ("code" OR "software
code" OR "source code" OR "system code")
```

### 3.1.4 Inclusion and Exclusion Criteria

The inclusion and exclusion criteria allow classifying each primary study as a candidate to be included or excluded from the SLR [Kitchenham and Charters, 2007]. As an SLR may involve many studies, we limited the scope of selecting only complete papers and ignoring short papers or documents classified as theses or dissertations. We decided to do so because the authors usually publish as full papers the studies regarding dissertations and theses. Besides, a short paper usually presents emerging results. It is essential to highlight that we consider full papers as being documents containing six pages or more. Table 3.3 presents the inclusion and exclusion criteria we have defined in this study.

Table 3.3: Inclusion and Exclusion Criteria.

| **Inclusion Criteria** |
|---|
| Papers published in English |
| Full papers |
| Papers published in Computer Science |
| Papers available in electronic format |
| Papers published in conferences and journals |
| Papers related to the topic investigated in this study |
| **Exclusion Criteria** |
| Duplicate studies |
| Documents classified as tutorials, posters, panels, talks, lectures, round tables, theses, dissertations, book chapters and technical report |
| Papers that cannot be found |

## 3.2 Execution

The execution phase consists of applying the search string to the electronic databases to identify candidate studies to be analyzed, as well as filtering the studies by using the inclusion and exclusion criteria to select only the relevant studies.

### 3.2.1 Search Process

We performed the search process from June $20^{th}$, 2019 to June $25^{th}$, 2019. We did not define any constraint during the search, and therefore, considered all studies returned by non-filter databases per publication year. Table 3.4 presents the results regarding the number of papers found in each electronic database. We obtained a total of 5,709 documents at the end of this process.

Table 3.4: Studies obtained after the search process.

| **Database** | **Studies Returned** |
|---|---|
| ACM Digital Library | 101 |
| Compendex (Engineering Village) | 246 |
| IEEE Xplore | 150 |
| Scopus | 5,117 |
| Web of Science | 95 |
| **Total** | **5,709** |

### 3.2.2 Papers Selection Process

As we identified a large number of 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 used as follows.

**Step 1 - Exclusion of duplicate studies.** In this step, we removed duplicate studies, ensuring that only one register of a given paper remains. To do so, we analyzed the title and authors of the documents and discarded the duplicate entries. This step removed 129 papers, resulting in 5,580 to Step 2 analyze.

**Step 2 - Exclusion of documents that are not papers.** It consisted of removing documents that not classified as complete papers. Therefore, this step discarded the ones classified as tutorials, posters, panels, lectures, round tables, theses, dissertations, book chapters, technical reports, and short papers. It is essential to highlight that we consider a complete paper as being the one with six pages or more. Articles under six pages were considered short papers and removed here. This step removed 411 documents, resulting in 5,169 studies to Step 3 analyze.

**Step 3 - Metadata Reading.** In this step, we analyzed the title and the abstract of the 5,169 papers obtained in Step 2 to select the ones relevant to this SLR. Although the papers' title and abstract provide an idea about the subject treated in the paper, it is sometimes necessary to make a more in-depth reading of the paper to

identify whether it is relevant. To avoid hasty decisions and exclude relevant documents, we classified as "dubious" the studies we were not able to select by reading the metadata. Step 4 analyzed such papers. At the end of Step 3, we identified 34 relevant papers already selected for this SLR, and 70 "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 SLR. To identify these occurrences, we performed a diagonal reading of the papers. The diagonal reading consisted of analyzing the introduction, topics, and conclusion of the papers to find more details. At the end of this step, we concluded that 26 out of the 70 "dubious" studies were relevant and therefore, we included them in this SLR.

**Step 5 - Snowballing.** Searching in the electronic database does not guarantee that all relevant studies related to a particular topic would 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 that are not found in the search process [Wohlin, 2014]. We may perform the snowballing process in two different 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 34 papers selected after Step 3, and 26 papers selected after Step 4. In total, we reviewed 2,104 references and selected 70 new studies in this step.

In summary, Steps 1 - 4 received as input the following quantity of primary studies, respectively: (i) 5,709; (ii) 5,580; (iii) 5,169; and (iv) 70 primary studies. At the end of Step 4, we identified 60 relevant studies being 34 after Step 3 and 26 after Step 4. Finally, Step 5 received 2,104 references from 60 papers already selected from the previous steps. This step recovered 70 studies, resulting in 130 papers selected for this SLR. Figure 3.1 summarizes the selection process. The selected primary studies were analyzed and summarized to answer the research questions

### 3.2.3 Data Extraction

We found 130 papers about the evolution of software structure and source code. The publication of these selected documents goes from 1979 to 2019. We read and summarized them to extract their primary information that answers the research questions. We made the data extracted from the primary studies selected to the SLR available on the website of our research group[1].

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

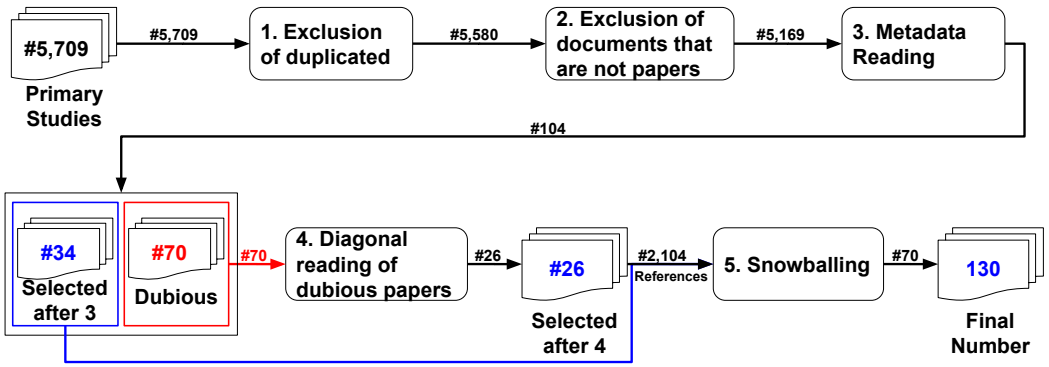


Figure 3.1: The filtering process carried out for selecting the primary studies.

## 3.3 Results

This section presents the results of this SLR. We report the results in seven sections according to the subject of the research questions investigated in this work:

- how the literature has approached research on software evolution;
- which characteristics of the datasets the studies on software evolution have used;
- details of studies that have investigated the applicability of Lehman's Laws;
- applications developed for software evolution analysis;
- studies on the evolution of quality attributes;
- studies on software structure evolution, and models for software evolution.

### 3.3.1 Approaches on Software Evolution

This section answers **RQ1:** *How has the literature approached studies on software evolution?*

With RQ1, we aim to identify the way the literature approached research on software evolution. To answer this research question, we analyzed each primary study's goal and classified them as some particular categories according to their content. We found five categories of studies on software evolution: (i) applicability of Lehman's laws; (ii) application; (iii) evolution of quality attributes; (iv) software structure evolution; and (v) model.

*Applicability of Lehman's laws* is a category that involves studies, which analyze if systems from different contexts have been developed to follow the laws of software evolution proposed by Lehman [1996]. We identify a total of 11 studies regarding this category. Lehman's laws, proposed in the mid '90s, are based on the evolution of large systems. The main objective of studies in this category is to check and validate the

presence of these laws in other systems and contexts and identify novel insights and systems' behavior over their evolution.

The *application* category consists of studies that propose and develop ways or environments, such as tools, strategies, techniques, plugins, among others, that support users' analysis and exploration of the evolution of software systems across multiple dimensions or versions. It comprises 34 studies and is the one that presented the most significant number of studies in this SLR. There are several input forms used by software evolution applications, such as source code, class diagram, and commit history. However, the source code is the most used input form.

The *evolution of the quality attributes* category covers studies investigating how internal quality attributes behave over the software systems' evolution. Internal quality attributes are properties, such as size, coupling, and cohesion, among others related to the development of the process or product [Sommerville, 2012]. This category is composed of 26 papers. Most studies of this category have analyzed the behavior of the quality attributes in terms of growth or decrease. Those studies' primary purpose is to identify alternatives and strategies that help developers improve the quality of the software systems.

The *software structure evolution* category consists of studies that explore the evolution of a particular dimension of the software system. Dimension refers to aspects such as architectural design, bad smells, code vulnerabilities, software quality, technical debt, among others. Out of the 130 papers identified in this SLR, 27 refer to this category.

The *model* category covers studies that analyze evolution data from particular properties of software systems and propose models to describe, represent, or predict how these properties evolve. Although there are some different software evolution models, several studies have proposed models to predict the defects in later software releases [Krishnan et al., 2011b; Ratzinger et al., 2007; Raja et al., 2009; Ohlsson et al., 2001; Khoshgoftaar et al., 1999; Arisholm and Briand, 2006; Shatnawi and Li, 2008]. Such models intend to allow developers and researchers to understand how software evolution occurs and serve as a powerful tool for monitoring possible problems that may arise in the software life cycle. This category consists of 32 studies and is the second in number of studies in this SLR.

**Summary of RQ1:** We conclude, in response to RQ1, that software evolution has been studied in five different approaches: (i) applicability of Lehman's laws, (ii) application, (iii) evolution of quality attributes, (iv) software structure evolution, and (v) model.

### 3.3.2 Software Evolution Datasets

This section answers **RQ2:** *What are the main features of the datasets used in studies on software evolution?*

RQ2 aims to identify and characterize the datasets the researchers have used to study software evolution. The response to this question is crucial as it can point out datasets that may be used in future studies and depict how diverse the samples considered are. We describe the datasets in three characteristics: (i) number of systems that compose them; (ii) type of dataset, i.e., if the dataset was proposed by the authors of the paper or the papers' authors used 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 3.2 and Table 3.5. By Figure 3.2 and Table 3.5, we conclude that the datasets are not composed of a large number of systems. We identified that, in the median, datasets had been composed of 1 to 6 systems, and in 75% of the cases, they presented not more than 14 systems. Regarding the *Application* and *Applicability of Lehman's Laws* categories, the largest dataset is composed of 23 systems.

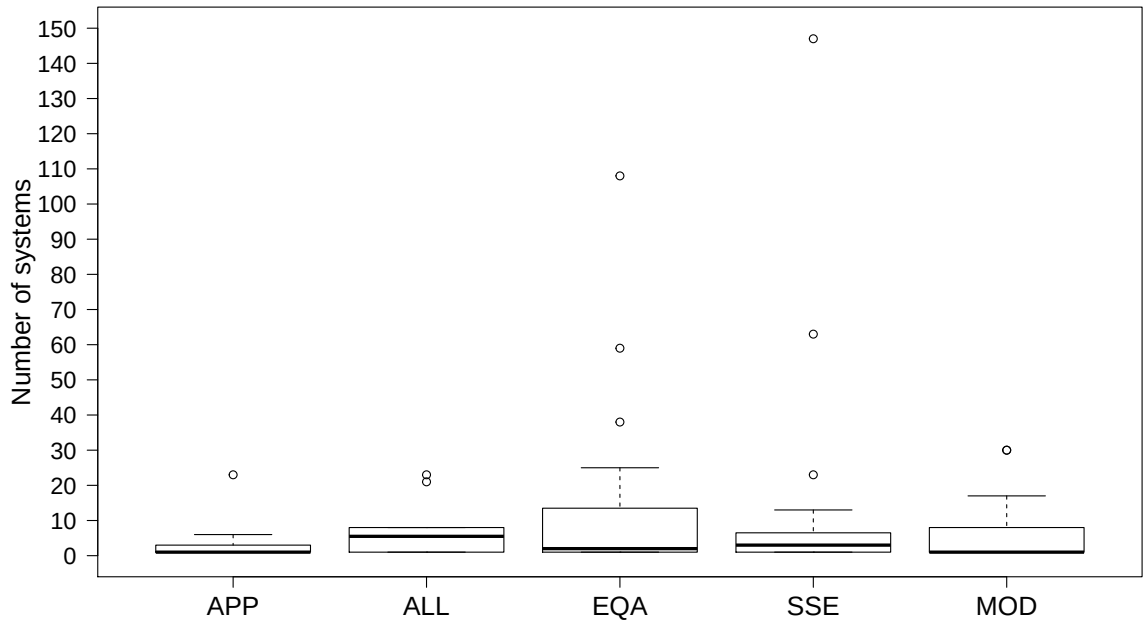


Figure 3.2: Distribution of the number of systems considered by the software evolution datasets. **APP**: Application; **ALL**: Applicability of Lehman's Laws; **EQA**: Evolution of Quality Attributes; **SSE**: Software Structure Evolution; **MOD**: Model.

Although our results show that, in general, the datasets on software evolution are not composed of a large number of systems, we also identified some outliers that

Table 3.5: Descriptive analysis of the distribution of the number of systems existing in the software evolution datasets.

| Category | 0% | 25% | 50% | 75% | 100% |
|---|---|---|---|---|---|
| Application | 1.00 | 1.00 | 1.00 | 3.00 | 23.00 |
| Applicability of Lehman's Laws | 1.00 | 1.75 | 5.50 | 7.75 | 23.00 |
| Evolution of Quality Attributes | 1.00 | 1.00 | 2.00 | 13.25 | 8,621.00 |
| Software Structure Evolution | 1.00 | 1.00 | 3.00 | 6.50 | 403,097.00 |
| Model | 1.00 | 1.00 | 1.00 | 8.00 | 8,621.00 |

indicate the existence of some datasets with a large sample of systems. For instance, in the *Evolution of Quality Attributes*, *Software structure Evolution*, and *Model* categories, we found datasets with 8,621 [Koch, 2007]; 403,097 [Kikas et al., 2017]; and 8,621 [Koch, 2005] systems, respectively. These values do not appear in the box plot chart in Figure 3.2, because we limited its y-axis at 150 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 found datasets, two of them [Koch, 2005, 2007] are not available online. However, the one proposed by Kikas et al. [2017] contains data about package dependency networks regarding the evolution of systems developed in JavaScript[2], Ruby[3], and Rust[4]. It is publicly available[5] for access and use by other studies.

We classified the dataset used by each study as "own" or "third-party", i.e., a dataset that was created by the respective paper's authors, and a dataset that was not created by the paper's authors. We identified that the vast majority of the studies used a dataset created by their authors, 115. Only ten studies used a third-party dataset. We also found five papers that did not mention any information about the origin of the dataset in their study. The "third-party" datasets found in this SLR are:

- SourceForge [Koch, 2005; Stewart et al., 2006; Koch, 2007];
- Apache ecosystem [Bavota et al., 2013; Digkas et al., 2017];
- GitHub [Kikas et al., 2017; Fernandez and Bergel, 2018];
- Helix [Yazdi et al., 2014];
- PROMISE [Alenezi and Zarour, 2015];

[2] `https://www.javascript.com/`
[3] `https://www.ruby-lang.org/`
[4] `https://www.rust-lang.org/`
[5] `https://github.com/riivo/package-dependency-networks`

- Qualitas Corpus [Singh and Ahmed, 2017]

Qualitas Corpus[6], PROMISE[7], and Helix[8] are software evolution repositories built to support studies on software evolution, while GitHub[9], SourceForge[10] and Apache ecosystem[11] 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 information by creating scripts or tools.

Finally, we analyzed the data time frame that the studies have 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 composed by year and month (YYYY-MM), we counted the monthly number of systems whose information refers to the respective date. For instance, suppose that 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. It is essential to highlight that we included only the time frame of studies, which specified year and month. We excluded from our analysis the time frame that did not follow this pattern. Then, in total, this analysis considered the systems' time frame from 47 out of the 130 primary studies identified in the SLR. Figure 3.3 summarizes our analysis by a heat map chart where the lines represent the months, and the columns, the years.

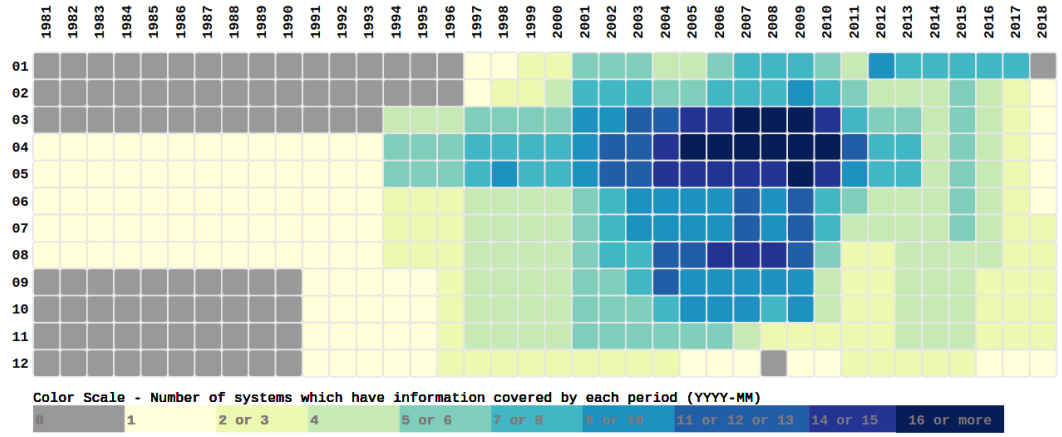


Figure 3.3: Number of systems considered in the studies by month/year.

[6] http://qualitascorpus.com/
[7] https://code.google.com/p/promisedata/
[8] http://www.ict.swin.edu.au/research/projects/helix/
[9] https://github.com/
[10] https://sourceforge.net/
[11] http://archive.apache.org/dist/

Analyzing Figure 3.3, we observe that the datasets consider data from 1981 to 2018. However, the period from 2001 up to 2010 concentrates most of the data collected from the systems. Although there are evolution data collected between 2016 and 2018, this period covers a low number of systems. Therefore, this finding reveals the need to build datasets with more current data on software evolution.

**Summary of RQ2:** 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. Finally, we identified that most of the datasets are composed by data collected from 2001 to 2010.

### 3.3.3 Applicability of Lehman's Laws

This section answers RQA.1 and RQA.2 based on the 11 studies found for this category.

**RQA.1:** *Which Lehman's Laws have been validated?*

RQA.1 aims to identify the laws, proposed by Lehman [1996], the ones considered in studies of software evolution. Figure 3.4 summarizes the number of studies that have considered each Lehman's Law. The graphic shows the number of studies that have found results, which validate the law, did not validate it, and the inconclusive ones. The graphic also indicates the number of studies that did not consider the respective law.

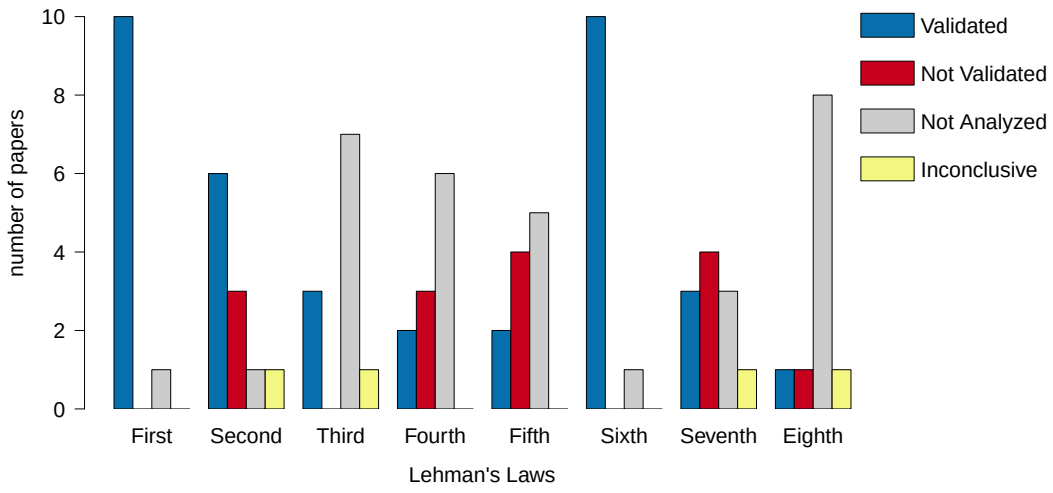


Figure 3.4: Frequency of validation of the Lehman's laws.

Analyzing Figure 3.4, we observed four Lehman's laws that have mostly attracted attention from researchers in the literature. They are:

1. continuing change (1st law)
2. increasing complexity (2nd law)
3. stability, conservation of familiarity (5th law)
4. continuing growth (6th law)
5. declining quality (7th law).

In contrast, the other four laws are less investigated. They are:

1. self-regulation (3rd law);
2. conservation of organizational (4th law)
3. feedback system (8th law).

1st Lehman's law indicates that systems need to be in progressively change and adaptation over the software evolution to be more satisfactory [Lehman, 1996]. The 6th Lehman's law conjectures that due to the need to maintain user satisfaction, a software system needs to grow along the software lifetime [Lehman, 1996]. Among the most investigated laws, the 1st and the 6th laws have been validated. All papers that analyzed these laws have confirmed them. Few studies have considered the 3rd Lehman's law; however, most of them confirmed its occurrence. This law indicates that the system evolution process is self-regulating, and the maintenance process controls the growth rate [Lehman, 1996].

The 2nd and 7th Lehman's laws state that unless a system has never been submitted to maintenance work, its complexity tends to increase, and its quality tends to decline over time [Lehman, 1996]. Regarding the 2nd law, although studies have pointed out that it tends to occur over time, there is some evidence that the occurrence of this law may be sensitive to some contexts. For instance, three studies analyzed in this SLR checked the occurrence of the 2nd Lehman's law in the evolution of mobile applications [Gezici et al., 2019; Zhang et al., 2013; Li et al., 2017], and none of them was able to confirm it. Besides, one of them compared the increasing complexity in two different contexts, mobile and desktop applications, and identified that the growth trends in these contexts are different. The studies analyzed in this SLR show results that contradict what the 7th law says since 44% of the works that considered that law have pointed out that the software quality does not tend to decline over time, and therefore, they did not confirm this law.

4th Lehman's law defines that the average global rate of activity in an evolving system is invariant over the product's lifetime [Lehman, 1996]. The 5th Lehman's law suggests that the familiarity with evolving software systems is conserved, i.e., their content of successive releases is not statistically invariant [Lehman, 1996]. Observing Figure 3.4, we may see that a large number of papers, around 55% and 45%, do not investigate these laws, respectively. This fact is due to the difficulty of measuring activity rate and content familiarity. The studies that considered those laws have not found evidence of their occurrences.

Finally, the 8th Lehman's law conjectures that the evolution process depends on feedback systems to achieve significant improvement [Lehman, 1996]. This law has not been much analyzed in the literature, and its occurrence is inconclusive. We may observe in Figure 3.4 that only three papers investigated it, and all of them disagreed with each other about the occurrence of this law into the software evolution process. Therefore, the investigation of this law is still open in the literature.

**Summary of RQA.1:** In response to RQA.1, we conclude that the literature has confirmed the 1st, 3rd, and 6th Lehman's laws. On the other hand, the results found so far do not confirm the 4th, 5th, and 7th laws. The 2nd law is partially supported since it seems to be sensitive to some applications context. Finally, the results of 8th Lehman's law are inconclusive, and its confirmation is open in the literature.

**RQA.2:** *In which application contexts the Lehman's laws have been analyzed?*

Table 3.6 shows the application contexts investigated in each study analyzed in this SLR. We identified five different types of contexts that investigated Lehman's laws. They are: (i) mobile application (MA); (ii) open-source software (OSS); (iii) eclipse third-party plugins (ETPP); (iv) proprietary software (PS); and (v) C library (CLIB).

In most cases, the studies about the Lehman's laws have chosen open-source systems in their analysis. A total of 45% of those studies have considered OSS projects, while 27% considered the mobile application of the papers. Just a small number of studies have considered the other three contexts. Each of them was investigated by only one paper, which corresponds to 9% of the sample.

**Summary of RQA.2:** Lehman's laws have been investigated in five contexts: (i) mobile application; (ii) open-source software; (iii) eclipse third-party plugins; (iv) proprietary software; and (v) C library. It also indicates that open-source software and mobile application are the most scenarios in the literature to investigate Lehman's laws.

Table 3.6: Contexts in which the Lehman's laws have been investigated.

| Paper[Ref.] | MA | OSS | ETPP | PS | CLIB |
|---|---|---|---|---|---|
| Gezici et al. [2019] | • | | | | |
| Israeli and Feitelson [2010] | | • | | | |
| Businge et al. [2010] | | | • | | |
| Alenezi and Almustafa [2015] | | • | | | |
| Barry et al. [2007] | | | | • | |
| Lee et al. [2007a] | | • | | | |
| Gonzalez-Barahona et al. [2014] | | | | | • |
| Mens et al. [2008] | | • | | | |
| Zhang et al. [2013] | • | | | | |
| Li et al. [2017] | • | | | | |
| Xie et al. [2009] | | • | | | |
| **Total** | 3 | 5 | 1 | 1 | 1 |

### 3.3.4 Application on Software Evolution

This section answers RQB.1 and RQB.2 based on the 34 studies found for this category.

**RQB.1:** *What are the software evolution applications proposed or used in literature?*

RQB.1 aims to identify the software evolution applications that the papers have proposed. Although we identified 34 works regarding the application category, two of them report the same application. Due to this, we identified a total of 33 applications and summarized them in Table 3.7 with their respective references.

We grouped the applications into two categories. The first one consists of techniques, approaches, or tools for online use, download, or replication. In this category, we found 16 applications. The second category consists of 17 applications proposed in the literature, but they are not available online for use or download. Although we identified a set of 33 applications, their authors did not define any name for six of them. In these cases, we labeled them as "Application name not mentioned".

**Summary of RQB.1:** We identified 33 applications, of which 16 are available online for use or download, and 17 are unavailable.

**RQB.2:** *What are the main features of these applications?*

RQB.2 investigates the software evolution applications' main features by considering their application types and the programming languages used for their development. Initially, we answer this research question by discussing the type of software evolution applications proposed in the literature. We identified six types: (i) visualization; (ii) technique; (iii) framework; (iv) metric; (v) plugin; and (vi) desktop tool. Figure 3.5

Table 3.7: List of software evolution applications found in this SLR.

| Applications Available Online or for Download |
|---|
| Dominance Relation [Burd and Munro, 1999]; Similarity [Merlo et al., 2002]; EvoLens [Ratzinger et al., 2005]; CVSscan [Voinea et al., 2005]; Churrasco [D'Ambros and Lanza, 2008]; Changing lines of code method (CLOC) [Baer and Zeidman, 2009]; egypt [Terceiro et al., 2009]; Chronos [Servant and Jones, 2012]; Product Evolution Tree [Kanda et al., 2013]; ETGM algorithm [Kpodjedo et al., 2013]; Mervin [Zoubek et al., 2018]; Application name not mentioned [Kemerer and Slaughter, 1999; Nikora and Munson, 2004; Xing and Stroulia, 2004; Stopford and Counsell, 2008; Sangwan et al., 2010] |
| Applications Proposed in Literature but Unavailable Online or for Download |
| Evolution Matrix [Lanza and Ducasse, 2002]; BEAGLE [Tu and Godfrey, 2002]; Gevol [Collberg et al., 2003]; Evolution Spectrograph [Wu et al., 2004b,a]; RelVis [Pinzger et al., 2005]; JDEvAn [Xing and Stroulia, 2005]; Evolution Storyboard [Beyer and Hassan, 2006]; Evolution Radar [D'Ambros et al., 2006]; EvoGraph [Fischer and Gall, 2006]; YARN [Hindle et al., 2007]; VERSO [Langelier et al., 2008]; Code Flows [Telea and Auber, 2008]; Replay [Hattori et al., 2013]; AniMatrix [Rufiange and Melancon, 2014]; IHVis [Rufiange and Fuhrman, 2014]; MultiPile [Fernandez and Bergel, 2018]; Application name not mentioned [Chevalier et al., 2007] |

ranks the types of the main features found in this SLR, considering the number of applications proposed for each of them.

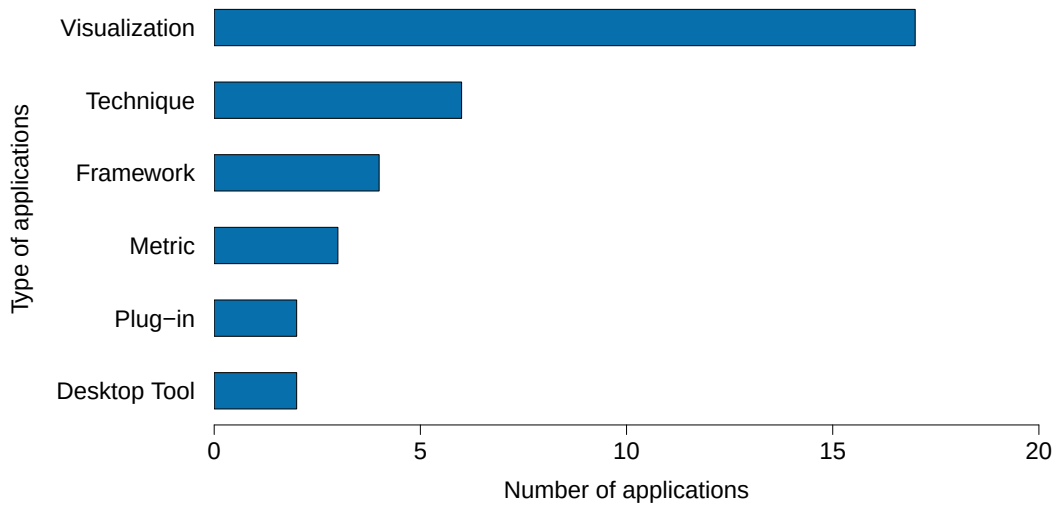


Figure 3.5: Types of applications that have been proposed in the literature.

Analyzing the available data from the 34 papers regarding the application category, we observed that 50% of the studies, 17 in total, proposed software visualization

tools. These tools aim to provide information about the internal structure or architecture of the software system via static, interactive, animated, or 3-D representations. In software evolution, the data have been extracted by versions, and representation has been projected for each of them. Such tools help monitor the changes in the software structure and help developers decide how to improve it. Another type of software evolution application proposed is technique. We identified that 17% of the papers, six in total, developed new techniques to support analysis and assessment in the evolution of a particular system. We consider as a technique in this study any approach, evaluation method, or algorithm proposed in the analyzed papers.

A framework is a type of application proposed in less quantity than software visualization tools and techniques. We found that 12% of the studies, four in total, developed frameworks to support software evolution. The studies have also proposed some metrics to help developers to measure the quality of evolution of software systems. We identified three works, 9% of the analyzed papers, that designed this type of application. The proposed metrics are (i) dominance relation, (ii) changing lines of code method (CLOC), and (iii) similarity. Dominance relation aims to assess the code maintainability from the software over time [Burd and Munro, 1999]. Changing lines of code method consists of analyzing lines of code that are modified, added, or remain constant over the software lifetime [Baer and Zeidman, 2009]. Similarity aims to quantify the similarities of code fragments between the software releases [Merlo et al., 2002].

The other two types of applications proposed in the literature are plugins and desktop tools. Plug-in is an application with some particular functionalities which run together with more extensive programs. Usually, the plugins have been developed to run with Eclipse[12]. The Desktop tool is an application that we may download, and in some cases, installs on a computer so that it may run locally. Figure 3.5 shows that 6% of the analyzed studies, two in total, produced plugins and desktop tools.

Furthermore, we investigated the programming languages the studies have used to develop the proposed applications. Figure 3.6 summarizes those programming languages, as well as the number of applications that used them. For the studies that proposed applications, but did not implement them, we labeled the programming languages as "Not mentioned" in Figure 3.6.

We observed that, in many cases, the studies propose applications, but do not implement them. We identified five programming languages used to develop the applications. In this sequence, Java and C++ have been predominant in the researchers'

[12] `https://www.eclipse.org/`

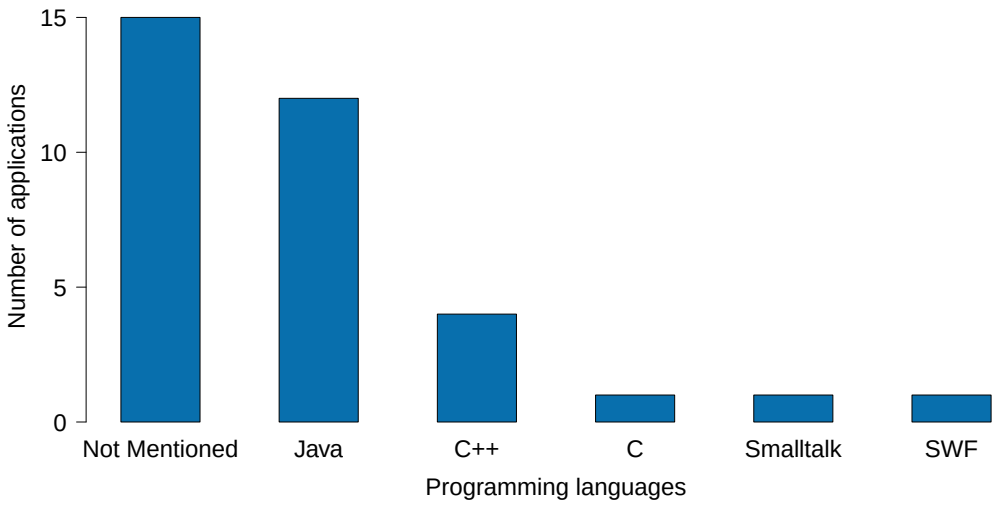


Figure 3.6: Programming languages used to develop software evolution applications.

preference and choice. The other three programming languages, C, Smalltalk, and SWF (Shockwave Flash), are less popular. SWF is a light file format for web applications that allow inserting multimedia content on websites [Schaeffer, 2009].

**Summary of RQB.2:** We identified six types of applications: visualization, technique, framework, metric, plugin, and desktop tool. The literature points out the software visualization tools as the most common type of application developed. Most of the studies proposed an application but did not implement it. Among the studies that proposed and implemented their applications, Java and C++ are the programming languages that have been most used by researchers to develop their applications.

### 3.3.5 Evolution of Quality Attributes

This section analyzes the evolution of quality attributes to answer RQC.1 and RQC.2 based on the 26 studies found for this category.

**RQC.1:** *Which quality attributes have been analyzed in studies on software evolution?*

As stated in Section 3.3.3, the systems continually change and adapt over their evolution to become more satisfactory and meet all the needs required by their users. 1st Lehman's law describes this behavior, and the studies have confirmed it. Because of this phenomenon, many studies on software evolution have concentrated on investi-

gating how the software quality has evolved via some properties denominated internal quality attributes. This investigation line is one of the most explored in software evolution literature. Therefore, our goal with RQC.1 is to identify all the internal quality attributes considered in studies on software evolution. We identified 14 internal quality attributes that have been studied in this context. Table 3.8 summarizes the findings.

As we may observe in Table 3.8, the evolution of systems' size (SI) is the mostl investigated, possibly due to the high variability of metrics available to measure this attribute, such as lines of code and number of files. These studies aim to understand how the system size behave over time and identify a pattern to characterize size evolution. All studies have found that systems' size grows over time, although there is still no conclusion regarding this growth pattern. For instance, Godfrey and Tu [2000]; Herraiz et al. [2006]; Koch [2007]; Gonzalez-Barahona et al. [2009] argue that open-source systems grow super-linearly, whereas Capiluppi and Ramil [2004]; Izurieta and Bieman [2006]; Capiluppi et al. [2007] indicate that both open-source and proprietary software follow a sub-linear growth. Robles et al. [2005] suggest that open-source software follow a linear growth pattern, and both super-linearity and sub-linearity occur exceptionally. Herraiz et al. [2007] identified that the distribution of size values over time follows a Pareto distribution and, then, this type of distribution may be a candidate method to represent the size evolution pattern of the software. Given that there is no clear growth pattern that explains the evolution of the size, future work should explore this subject more.

Complexity (CP) has also been widely investigated in the literature. Studies regarding the complexity evolution have diverged about the complexity growth pattern. For instance, Antinyan et al. [2013] indicated that the complexity increases fastly, whereas Stewart et al. [2006] found that software complexity decreases as the software grows in size. Some studies have aimed to identify these divergences analyzing factors that impact the complexity of growth or decrease to explain these two phenomena. Stewart et al. [2006]; Terceiro et al. [2012]; Capiluppi and Ramil [2004]; Herraiz et al. [2007] identified the existence of a relationship between the size and complexity attributes. Consequently, the system growth in size directly impacts the complexity increase or decrease over time. Terceiro et al. [2012] also present evidence that other variables, such as developer experience and change diffusion, affect the complexity growth over time. Besides, concerning the relationship between size and complexity, Capiluppi et al. [2007] found that the use of the agile method in the software development process smoothes the evolution, and avoids problems of increasing complexity over the lifetime.

Some internal quality attributes had a little investigation in the literature of

Table 3.8: Summary of internal quality attributes extracted from the papers regarding the evolution of the quality attributes category.

| Paper[Ref.] | CH | CO | ST | MD | CP | SI | DS | ACT | ARW | SE | IH | DW | RE | MT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Singh and Ahmed [2017] | • | • | | | | | | | | | | | | |
| Yu and Ramaswamy [2009] | | | • | | | | | | | | | | | |
| Alenezi and Zarour [2015] | | | | • | | | | | | | | | | |
| Antinyan et al. [2013] | | | | | • | | | | | | | | | |
| Grigorio et al. [2015] | | | | | • | | | | | | | | | |
| Stewart et al. [2006] | | | | | • | | | | | | | | | |
| Capiluppi et al. [2004b] | | | | | | • | • | | | | | | | |
| Thomas et al. [2014] | • | | | | | | | | | | | | | |
| Capiluppi and Ramil [2004] | | | | | • | • | | • | • | | | | | |
| Al-Ajlan [2009] | | | | | | | | | | • | | | | |
| Terceiro et al. [2012] | | | | | • | | | | | | | | | |
| Nasseri et al. [2008] | | | | | | | | | | | • | | | |
| Capiluppi et al. [2007] | | | | | • | • | | | | | | | | |
| Vasa et al. [2009] | | | | | | | | | | | | • | | |
| Herraiz et al. [2006] | | | | | | • | | | | | | | | |
| Krishnan et al. [2011a] | | | | | | | | | | | | | • | |
| Robles et al. [2005] | | | | | | • | | | | | | | | |
| Godfrey and Tu [2000] | | | | | | • | | | | | | | | |
| Darcy et al. [2010] | | | | | • | • | | | | | | | | |
| Thomas et al. [2009] | | • | | | | | | | | | | | | • |
| Gonzalez-Barahona et al. [2009] | | | | | | • | | | | | | | | |
| Koch [2007] | | | | | | • | | | | | | | | |
| Izurieta and Bieman [2006] | | | | | | • | | | | | | | | |
| Capiluppi et al. [2004a] | | | | | | • | | | | | | | | |
| Hatton et al. [2017] | | | | | | • | | | | | | | | |
| Herraiz et al. [2007] | | | | | • | • | | | | | | | | |
| **Total** | 2 | 2 | 1 | 1 | 8 | 13 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**CH**: change; **CO**: coupling; **ST**: stability; **MO**: modularity; **CP**: complexity; **SI**: size; **DS**: directory structure; **ACT**: activity; **ARW**: anti-regressive work; **SE**: software evolution in general; **IH**: inheritance hierarchy; **DW**: distribution of wealth; **RE**: reliability; **MT**: maintainability.

software evolution. We found no more than two works that have studied the evolution of each of these attributes; Table 3.8 shows them. In the following we describe their central insights. Singh and Ahmed [2017] identified that coupling (CP) and change (CH) are correlated over time, and classes with high coupling in systems may be less fault-prone and may not lead to more changes. They consider change as the likelihood of this event occur in parts of a software system and characterize this aspect by using public interface change (PIC) and implementation changes (IMC) metrics. Thomas et al. [2014] also investigated change and found that there are some ways to analyze this property over time via change activities, such as corrective evolution, internal improvements, and new features. Thomas et al. [2009] investigated the evolution of coupling (CO) in the Linux system and if this attribute impacts the maintainability (MT) evolution. They concluded that coupling grows linearly, and its growth does not affect the Linux maintainability. Yu and Ramaswamy [2009] pointed out that the evolution of software stability (ST) is more affected by the types of functions existing in the software than by the system environment. Alenezi and Zarour [2015] investigated the evolution of modularity (MD) considering two systems and identified evidence that this quality attribute does not improve over time. Capiluppi et al. [2004b] analyzed the evolution of directory structure (DS) and found that the number of folders has an increasing trend, and the number of files per level increases continuously. They also identified that the depth of the directories tree tends to stabilize over time, whereas the directories tree's width tends to increase. Capiluppi and Ramil [2004] investigated the evolution of activity (ACT) and anti-regressive work (ARW). They define activity as the support and modifications that the systems receive from their developers and maintainers. Anti-regressive work is the portion of the software which has gone through refactoring activities to reduce the complexity. They identified that both activity and anti-regressive work attributes present ripples and cyclic trends.

Al-Ajlan [2009] studied the evolution of Eclipse metrics in open-source software. Since the author did not specify any attribute, we called their analysis software evolution in general (SE). Al-Ajlan [2009] concluded that software systems grow in modules and lines of code. Nasseri et al. [2008] investigated the evolution of inheritance hierarchy (IH) and identified a strong tendency of classes to be added at the levels 1 and 2 of the hierarchy. Vasa et al. [2009] analyzed the distribution of wealth (DW) over time. They defined wealth as high concentrations of values in specific locations of the system. The distribution of wealth consists of a comparison of metric values regarding several components of the software. They identified that the distribution of wealth, in terms of the Gini coefficient, changes little between subsequent releases. Krishnan et al. [2011a] studied the evolution of reliability (RE) in the software product line. In

their work, they define reliability as continuity of correct service. They were not able to identify if the reliability has improved as the product line matures. However, they found that the reliability may be impaired when on-going change stays higher than commonly supposed in software. Finally, Merlo et al. [2002] examined the evolution of similarity (SM) in the Linux Kernel, in terms of code fragments or modules. They describe the similarity as similar code fragments shared by different systems.

**Summary of RQC.1:** We identified 14 quality attributes studied in the context of software evolution. Table 3.8 summarizes all identified quality attributes and shows the papers that investigated each one of them. Although we identified a large number of quality attributes that have been investigated in studies on software evolution, little is known about the pattern of evolution of these properties and how they evolve.

**RQC.2:** *Which software metrics have been considered to analyze the evolution of software quality attributes?*

With RQC.2, we aim to compile the software metrics that have been used to measure and analyze the evolution of the quality attributes identified in RQC.1. For this purpose, we identified the metrics described in the works that considered the evolution of quality attributes. We identified a set of 72 software metrics and summarized them in Tables A.1 and A.2, located in the Appendix of this paper. Figure 3.7 shows the mapping between quality attributes and software metrics that we found. The cases in which the mapping between an attribute of quality and a software metric has been filled with gray in the table indicate that none of the analyzed papers reported a metric for that attribute. On the other hand, if the mapping between an attribute of quality and a software metric is filled with another color, the papers pointed relation between them. The more to the right the color is in the color scale, the higher the number of papers that reported the mapping.

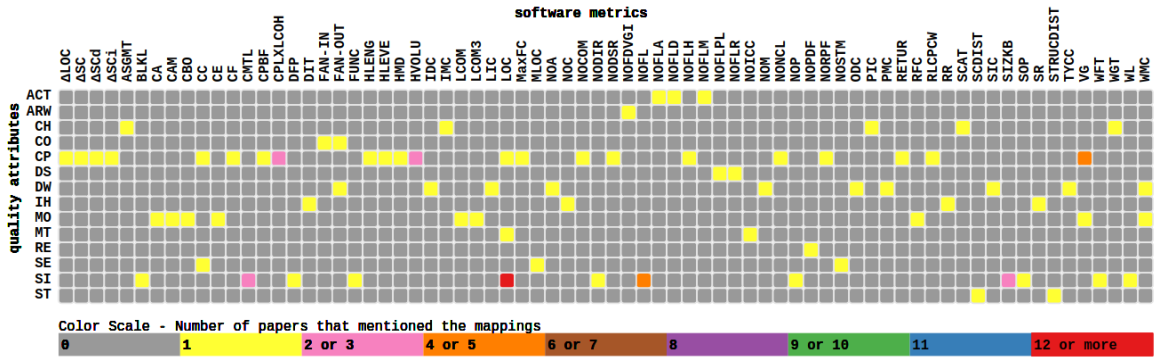


Figure 3.7: Mapping between software metrics and quality attributes.

As shown in Figure 3.7, there is a large quantity and diversity of metrics used to measure the quality attributes in software evolution works. Among all quality attributes, complexity (CP) was the one that presented the most substantial quantity of software metrics. We identified 22 software metrics for it. Size (SI), distribution of wealth (DW), and modularity (MO) have been measured by 12, 10, and 9 metrics.

In general, the studies define their own metrics to measure the quality attributes over time. Only one primary study mentioned most of the relations between quality attributes and metrics. Figure 3.7 also reveals patterns regarding the use of some software metrics of size and complexity quality attributes. The lines of code (LOC) is the most used metric to measure the size in studies on software evolution. The number of files (NOFL) is another essential size metric, but it is less used than LOC. We also identified that number of comment lines (CMTL) and total size in KB (SIZKB) are size metrics, which have been used as a complement to LOC and NOFL. Regarding complexity, the primary studies have used McCabe's complexity (VG) as the primary complexity metric. CplXLCoh and Halstead's volume (HVOLU) had been rarely used.

**Summary of RQC.2:** We identified 72 software metrics used in the literature to analyze internal quality attributes' evolution. Tables A.1 and A.2 show the complete set of metrics we found, and Figure 3.7 represents the mapping between the software metrics and the quality attributes. Some popular metrics, such as lines of code and number of files, frequently appear in studies on software evolution. However, those works do not present a pattern for measuring internal quality attributes.

### 3.3.6 Software Structure Evolution

This section analyzes how software structure evolution has been studied, answering RQD.1 and RQD.2 based on the 27 studies found for this category.

**RQD.1:** *Which dimensions of software structure have been evaluated in the literature?*

RQD.1 consists of identifying the dimensions related to software structure analyzed by studies on software evolution. Table 3.9 summarizes the dimensions we identified in this SLR, followed by the number of studies in parentheses that investigated them, and the reference of each work.

We identified ten subjects in the context of software evolution. The studies have often chosen to investigate the evolution of systems' internal structure, the occurrence

Table 3.9: Software structure dimensions investigated in studies about software structure evolution.

| Software Structure Dimensions |
|---|
| Internal Structure (8) [Gall et al., 1997; Barry et al., 2003; Li et al., 2008; Wang et al., 2009; Savić et al., 2011; Wang et al., 2012, 2013; Kikas et al., 2017]; Bad smell (7) [Antoniol et al., 2002; Olbrich et al., 2010; Saha et al., 2013; Chatzigeorgiou and Manakos, 2014; Rani and Chhabra, 2017; Kanwal et al., 2018, 2019]; Architectural design (5) [Tahvildari et al., 1999; Ferreira et al., 2012; German et al., 2013; Decan et al., 2019; Spinellis and Avgeriou, 2019]; Co-evolution between types of coupling (1) [Yu, 2007]; Faults (1) [Ostrand and Weyuker, 2002]; Function side effects (1) [Alnaeli et al., 2016]; Internal quality (1) [Longo et al., 2008]; Project inter-dependencies (1) [Bavota et al., 2013]; Technical debt (1) [Digkas et al., 2017]; Vulnerabilities (1) [Massacci et al., 2011] |

of bad smells, and architectural design. In total, eight, seven, and five papers report results about these three dimensions, respectively. For the other seven dimensions, we found only one paper that analyzed each one of them.

**Summary of RQD.1:** The literature has studied the software structure evolution via ten dimensions of software structure. The dimensions more frequently investigated are the internal structure, bad smell occurrence, and architectural design. Co-evolution between types of coupling, faults, function side effects, internal quality, project inter-dependencies, technical debt, and vulnerabilities have also been considered, however, on a smaller scale.

**RQD.2:** *What are the main insights reported in the literature regarding software structure evolution?*

RQD.2 investigates the main findings reported in the literature regarding software structure evolution. To organize our discussion, we grouped the papers according to the dimension they consider.

Most papers have investigated the evolution of the internal organization of software systems. Gall et al. [1997] analyzed the internal structure of proprietary software over its historical development and found that the system's internal structure and its subsystems differ over time. Barry et al. [2003] analyzed the internal structure of 23 systems and identified four groups of volatility patterns that they tend to follow over their lifetime. They consider volatility as a multi-dimensional phenomenon, which involves:

- change in the software size (amplitude)

- frequency with which systems are changed (periodicity), and
- variation of the system behavior considering amplitude and periodicity (dispersion).

They concluded that not all systems follow the same volatility patterns. Kikas et al. [2017] studied the package dependency networks of some ecosystems to understand their main characteristics over the evolution. They concluded that the analyzed ecosystems are alive and growing. They also found that the ecosystems have become less dependent on a single popular package over time, and the vulnerability to the removal of the most popular package is increasing.

Moreover, several studies have identified some rules about the evolution of the internal software structure using complex networks theory. Those studies have concluded that the evolution of open-source software projects follows scale-free and small-world properties [Li et al., 2008; Wang et al., 2009; Savić et al., 2011; Wang et al., 2012, 2013]. The preferential attachment property has also been pointed out as a feature followed by these systems and a valuable property to explain how the networks evolve into a scale-free state [Wang et al., 2009; Savić et al., 2011; Wang et al., 2013]. Besides, Li et al. [2008] identified the other two essential rules. They found that the growth of nodes and links in the object-oriented software network is harmonic, and the clustering coefficient of late and old nodes are close over time.

Bad smells are symptoms existing in the structure or source code of a system that reflects poor design quality [Fowler et al., 1999]. The studies have analyzed how some types of bad smells have evolved over the software lifetime and extracted insights about their evolution. Olbrich et al. [2010] identified that the occurrences of the God Class and Brain Class bad smells over the software evolution have a negative effect regarding change and defects. They also found that God Class has a significant and positive correlation with the system size, and the emergence of this smell becomes more likely when a system grows. Rani and Chhabra [2017], and Chatzigeorgiou and Manakos [2014] analyzed the impact of Feature Envy, Switch Statements, Long Method, and God Class bad smells on the software evolution. In both studies, the authors concluded that these bad smells increase over the system evolution, and they accumulate as the system matures. Chatzigeorgiou and Manakos [2014] also found that these bad smells disappear because of the side effects of adaptive maintenance and not due to targeted refactoring activities. Antoniol et al. [2002] analyzed the evolution of Duplicated Code in the Linux kernel and concluded that this bad smell has not increased and deteriorated the Linux structure. Kanwal et al. [2018, 2019] studied and compared the evolution of structural clones and simple clones. According to them, simple clones are textual of

syntactically similar code fragments, whereas structural clones are recurring patterns of simple clones in software systems. Besides, although structural clones frequently change less than simple clones, structural clones have more inconsistent changes. Therefore, structural clones require more intelligent management than simple clones.

Some papers had been based on a high level of systems granularity, such as architectural design, to extract new insights about their evolution. Tahvildari et al. [1999] investigated the best way to characterize the evolution patterns of open-source systems at the architectural level. They classified the evolution patterns in three categories: interface evolution, implementation evolution, and structure evolution. They concluded that these three patterns had been better characterized by using software metrics. Fan-in and fan-out are the metrics that best characterize the evolution of an interface. McCabe's complexity, Halstead effort, and the number of comments best characterize implementation evolution. The overall distance best characterizes software structure evolution. Ferreira et al. [2012] evaluated the evolution of software systems in a macroscopic view using the Little House model. They identified that classes within two components of Little House, called LSCC and Out, suffer substantial degradation over evolution. German et al. [2013] studied the evolution of the R ecosystem's core and user contributions packages. They concluded that:

**(i)** R ecosystem is growing super-linearly over time with a robust set of core packages

**(ii)** user-contributed packages typically are smaller and contain less documentation than core ones

**(iii)** the size of a package remains stable over time, and

**(iv)** the packages have few dependencies.

In the same line, Decan et al. [2019] detailed the evolution of package dependency networks regarding some relevant ecosystems. They identified that the networks tend to grow over time both in size and in the number of package updates. They also found that complexity does not decrease and tends to remain stable or increase over time. Spinellis and Avgeriou [2019] studied the evolution of Unix from a software architecture perspective. They identified that Unix growth, in terms of size, has happened uniformly. Regarding complexity, they found that Unix grew at the beginning of its development, but reduced and controlled over its lifetime.

Finally, studies have investigated other dimensions, such as faults, technical debt, internal quality, among others, to identify novel patterns and insights about software evolution. For instance, Yu [2007] studied the correlation between evolutionary and

reference couplings, and he concluded that there is a linear correlation between these two types of coupling. Ostrand and Weyuker [2002] investigated the distribution of faults over the evolution of an extensive industrial software system. They found that faults concentrate in a small number of files and a small percentage of code mass, and as the system evolves, the faults become increasingly concentrated in smaller portions of the code. They also identified a small number of post-releases that presented faults over the system evolution. Alnaeli et al. [2016] examined the prevalence and distribution of function side effects over software evolution to determine the main factors that have caused this problem. The authors concluded that modification of global variables and passing parameters by reference are the main actions that have resulted in side effects in functions. Longo et al. [2008] investigated how software quality has been impacted by corrective evolution, refactoring, and new functionalities. They found that corrective evolution, the addition of new functionalities, and one type of refactoring called coding convention adoption have not impacted the quality of the systems. On the other hand, another type of refactoring called adoption of new frameworks and libraries is the factory that has a significant impact on the software's internal quality. Bavota et al. [2013] observed some ecosystems' evolution to analyze how the project inter-dependencies behave and change over time. They identified that the projects and their dependencies increase continuously, and the dependencies follow different trends. Besides, they found that some projects tend to upgrade their dependencies when some critical changes are released, and the impact of the upgrade is usually low. Digkas et al. [2017] analyzed the evolution of technical debt in some open-source projects and concluded that the technical debt has increased with the growth of the size, the number of issues, and complexity metrics. They also identified that some frequent and time-consuming types of technical debt are related to improper exception handling and code duplication. Massacci et al. [2011] investigated the interplay of the source code from Firefox and some known vulnerabilities over time. They identified that many vulnerabilities of Firefox versions had been discovered when people in their after-life well use these versions. They also found that a possible explanation to the after-life vulnerabilities in Firefox is slow code evolution, that is, much code retained between released versions.

**Summary of RQD.2:** We compiled the main insights the studies have found about the software structure evolution. For a better understanding, we guided our discussion by considering the dimensions identified in RQD.1.

### 3.3.7 Model

This section analyzes the papers regarding the model category to answer RQE.1, RQE.2, RQE.3, and RQE.4 based on the 32 studies found for this category.

**RQE.1:** *What are the types of models on software evolution proposed in the literature?*

To answer this research question, we initially analyzed the papers presenting models to identify the proposed models and their primary goals and features. We identified the following types of software evolution models: (i) characterization; (ii) descriptive; (iii) prediction; and (iv) simulation. Figure 3.8 ranks the types of software evolution models identified in this SLR, considering the number of papers obtained for each one of them.

*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. It contains eight studies in total and was the one that presented the second largest number of studies. The structure of this type of model is composed of unambiguous components, which may provide a sufficiently precise representation of particular properties and their relationship. For instance, some software properties that are 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.

*Descriptive*: it consists of a category that aims to describe, understand, or explain how internal phenomena or events regarding the system behave over time. It contains six studies in total and was the one that presented the third largest number of studies. Besides, it allows developers and researchers to characterize the evolution pattern of systems and extract properties or insights that may help them to improve the structure and quality of the systems. 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 system event in the future. These models consider the evolution pattern of one or more predictor variables in trying to determine the probability of some phenomena occurs, such as the presence of defects, increasing system size, co-change, among others. This category is the one that presented the most number of studies, 15 in total.

*Simulation*: this category consists of mathematical business models that try to reproduce a real-life system via software by combining mathematical and logic concepts. The simulation model contains entities and activities. The entities refer to roles played

within the model, such as developers, software modules, and machines. The activities consist of tasks, such as processing, code implementation, and refactoring. This type of model has been very little explored in the literature. In this SLR, we identified only three studies published in sequence by the same authors. In the first studies, they used simulation methods and techniques to compare the size evolution between open-source and proprietary software and understand how the main characteristics from the evolution pattern in these two contexts differ over time [Smith et al., 2004, 2005]. Later, they proposed an agent-based simulation model to study how software properties, such as size, complexity, and effort, relate to each other over the evolution of open-source software systems [Smith et al., 2006].

**Summary of RQE.1:** We identify four types of models on software evolution. They are: (i) characterization; (ii) descriptive; (iii) prediction; and (iv) simulation.

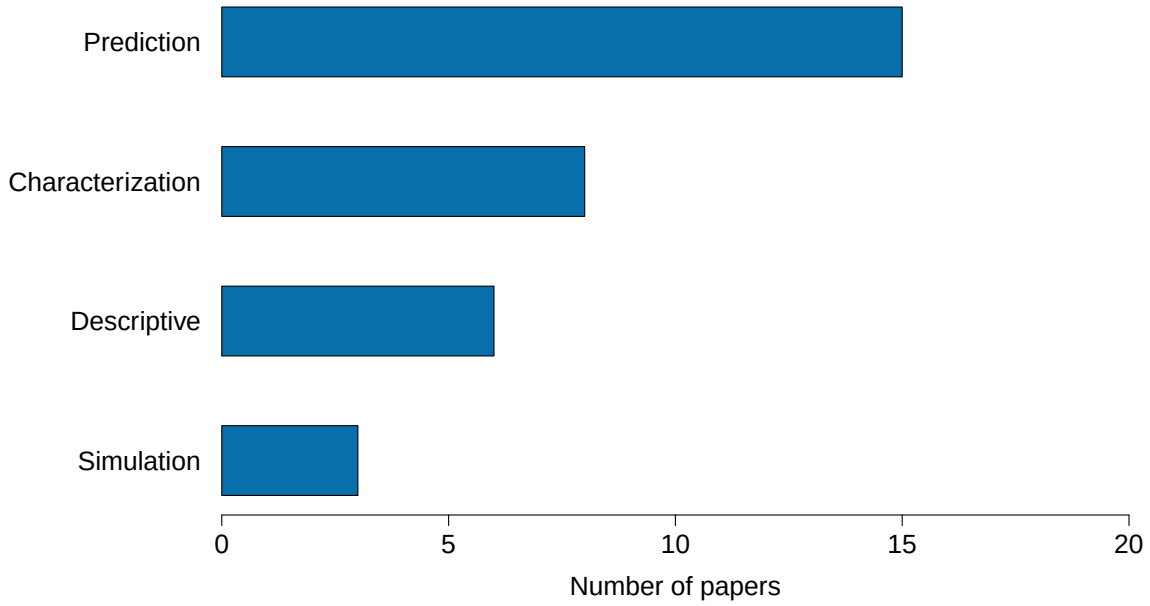


Figure 3.8: Number of papers by type of model.

**RQE.2:** *Which techniques have been used in the models of software evolution?*

Several approaches have been used to build prediction models, which we summarized in Table 3.10. We detected seven techniques: (i) regression; (ii) ARIMA; (iii) machine learning; (iv) Bayesian networks; (v) principal component analysis (PCA); (vi) Markov chain; and (vii) CART algorithm. Although they have been employed with the same purpose, there are some differences between them. For instance, regression techniques and ARIMA are strategies that have been most used to design

Table 3.10: Techniques used for designing the models on software evolution

| Paper[Ref.] | Type of model | Used techniques |
|---|---|---|
| Khoshgoftaar et al. [1999] | Prediction | CART algorithm |
| Graves et al. [2000] | | Regression techniques |
| Ramil and Lehman [2000] | | Regression techniques |
| Antoniol et al. [2001] | | ARIMA |
| Caprio et al. [2001] | | ARIMA |
| Ohlsson et al. [2001] | | Principal component analysis (PCA) |
| Arisholm and Briand [2006] | | Regression techniques |
| Ratzinger et al. [2007] | | Regression techniques and machine learning algorithm |
| Shatnawi and Li [2008] | | Regression techniques |
| Zhou et al. [2008] | | Bayesian network |
| Raja et al. [2009] | | ARIMA |
| Krishnan et al. [2011b] | | Machine learning algorithm |
| Yazdi et al. [2014] | | ARMA |
| Chaikalis and Chatzigeorgiou [2015] | | Complex networks |
| Trindade et al. [2017] | | Markov chain |
| Chen et al. [2008] | Characterization | Complex networks |
| Zheng et al. [2008] | | Complex networks |
| Ferreira et al. [2011] | | Complex networks |
| Pan et al. [2011] | | Complex networks |
| Bhattacharya et al. [2012] | | Graph-based model |
| Li et al. [2013] | | Complex networks |
| Wang et al. [2014] | | Complex networks |
| Liu and Ai [2016] | | Complex networks |
| Woodside [1979] | Descriptive | Mathematical methods |
| Capiluppi [2003] | | Regression techniques |
| Koch [2005] | | Regression techniques |
| Turnu et al. [2011] | | Yule process |
| Feitelson [2012] | | Not mentioned |
| Kirbas et al. [2014] | | Regression techniques |
| Smith et al. [2004] | Simulation | Qualitative reasoning technique |
| Smith et al. [2005] | | Qualitative reasoning technique |
| Smith et al. [2006] | | NetLogo multi-agent simulation tool |

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, whereas regression techniques already allow using one or more independent variables to predict future values for a different dependent variable. It is important to highlight that ARMA, reported in Table 3.10, is a variation of ARIMA methodology. Machine learning algorithms, specially tree-based and classifier algorithms, such as J48, C4.5, and M5, are other types of techniques that also have been applied to design prediction models. According to Ratzinger et al. [2007] and Krishnan et al. [2011b], these algorithms are very efficient in defining excellent and relevant predictors to the proposed models.

Bayesian networks, CART algorithm, complex networks, Markov chain, and principal component analysis (PCA) have also been used as alternative 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. PCA consists of a mathematical procedure that helps to reduce complex datasets with a high number of correlated variables to identify the central relationship between the variables [Kachigan, 1991]. 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 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. CART algorithm aims to build a parsimonious tree from continuous, ordinal predictors by first building the maximal tree, and after that, performs a pruning in it to obtain an appropriate level of detail [Breiman, 2017]. Complex networks consist of using the graph concepts to model the software structure as a network in which the nodes represent the internal software components, and 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 we 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 the assessment of 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.

Mathematical methods, regression techniques, and the Yule process have defined Descriptive models. Yule process is a technique that allows modeling many phenomena by using the Yule-Simon as the limiting distribution [Mitzenmacher, 2004]. Although they differ from each other regarding how they have been applied, the authors point out that these strategies are efficient for extracting unambiguous patterns that characterize the evolution of a given event of a software system.

Finally, qualitative reasoning techniques have supported the definition of Simulation models. Qualitative reasoning (QR) is a branch of artificial intelligence that automates reasoning about continuous aspects of the physical world to solve problems by using qualitative instead of quantitative information [Bredeweg and Struss, 2003]. The primary purpose of the techniques based on qualitative reasoning is to

apply representation and reasoning methods that allow computer programs to learn about physical systems' behavior without qualitative information. Besides, a NetLogo multi-agent simulation tool was also used by one of the studies when its purpose was to build an agent-based simulation model.

**Summary of RQE.2:** Table 3.10 summarizes the identified techniques we have found to build models on software evolution for each category of model. As the main strategies, we highlight the use of regression techniques and ARIMA for building Prediction models. Regression techniques are also relevant for producing Descriptive models. Graph-based techniques have been used in the Characterization model with an emphasis on complex networks. Finally, Simulation models have been defined by qualitative reasoning techniques.

**RQE.3:** *Which metrics have been defined in the literature to assess the accuracy of software evolution models?*

Some metrics have been used to measure the quality of the models' parameters and how effective the models are. We identified 29 assessment metrics. Table 3.11 reports these metrics by type of model.

In Table 3.11, we show references following each metric to indicate the papers that have chosen each of them and display the metrics that have been more employed to evaluate the type of model. Besides, we also included a label "Not mentioned", followed by references of papers that did not indicate the use of any assessment metric to evaluate their proposed models. In the Simulation models, we did not identify any type of assessment metric since the studies did not report any. For the Characterization model, although many papers did not report the metrics used in the study, we still identified one type of metric. On the other hand, the studies have used a large variety of metrics to assess the accuracy of Prediction and Descriptive models. Among the ones presented in Table 3.11, we may point out some of them with significant relevance since they have been chosen by more than one paper. They are: `R-square`, `Akaike information criterion (AIC)`, `false-positive rate (FPR)`, `mean absolute percentage error (MAPE)`, `mean squared error (MSE)`, and `recall`. Considering Prediction and Description models, `R-squared` is the most used metric. Four papers chose it to assess the models. Three papers mentioned `Akaike information criterion (AIC)`, and two papers used the metrics as mentioned above.

Table 3.11: Assessment metrics used to evaluate the accuracy of the models on software evolution.

| **Prediction** |
|---|
| Akaike information criterion (AIC) [Antoniol et al., 2001; Caprio et al., 2001; Yazdi et al., 2014]; False-positive rate (FPR) [Krishnan et al., 2011b; Arisholm and Briand, 2006]; Mean absolute percentage error (MAPE) [Caprio et al., 2001; Raja et al., 2009]; Mean squared error (MSE) [Ratzinger et al., 2007; Raja et al., 2009]; Recall [Zhou et al., 2008; Krishnan et al., 2011b]; Absolute prediction error (APE) [Caprio et al., 2001]; Classification-rule parameter [Khoshgoftaar et al., 1999]; Correlation coefficient (CC) [Ratzinger et al., 2007]; Error measure [Graves et al., 2000]; F-measure [Zhou et al., 2008]; Likelihood ratio test (LRT) [Shatnawi and Li, 2008]; Mean absolute deviation (MAD) [Raja et al., 2009]; Mean absolute error (MAE) [Ratzinger et al., 2007]; Mean magnitude of relative error (MMRE) [Ramil and Lehman, 2000]; Median magnitude of relative error (MdMRE) [Ramil and Lehman, 2000]; Misclassification rate [Khoshgoftaar et al., 1999]; Normalized mean squared error (NMSE) [Yazdi et al., 2014]; Normalized relative error (NRE) [Yazdi et al., 2014]; Observations with MMRE equal or lower than 10% (PRED(10)) [Ramil and Lehman, 2000]; Observations with MMRE equal or lower than 25% (PRED(25)) [Ramil and Lehman, 2000]; Percentage of correctly classified instances or accuracy (PC) [Krishnan et al., 2011b]; Percentage of false-negative [Arisholm and Briand, 2006]; Percentage prediction error (PPE) [Antoniol et al., 2001]; Precision Zhou et al. [2008]; Prediction error of estimation [Trindade et al., 2017]; R-squared [Ohlsson et al., 2001]; Not mentioned [Chaikalis and Chatzigeorgiou, 2015] |
| **Characterization** |
| Power law distributions [Li et al., 2013]; Not mentioned [Chen et al., 2008; Zheng et al., 2008; Ferreira et al., 2011; Pan et al., 2011; Bhattacharya et al., 2012; Wang et al., 2014; Liu and Ai, 2016] |
| **Descriptive** |
| R-squared [Capiluppi, 2003; Koch, 2005; Kirbas et al., 2014]; Estimated Yule distribution [Turnu et al., 2011]; Maximum likelihood (MLE) [Turnu et al., 2011]; Not mentioned [Woodside, 1979; Feitelson, 2012] |
| **Simulation** |
| Not mentioned Smith et al. [2004, 2005, 2006] |

**Summary of RQE.3:** Table 3.11 presents the 29 assessment metrics that the studies have used to evaluate the models. For Simulation models, we did not identify assessment metrics, and for Characterization models, we found only one type of metric. For Prediction and Descriptive metrics, we found a large number of metrics. The most used are: `R-square`, `Akaike information criterion (AIC)`, `false-positive rate (FPR)`, `mean absolute percentage error (MAPE)`, `mean squared error (MSE)`, and `recall`.

**RQE.4:** *Which software aspects have been considered in software evolution models?*

Table 3.12: Object of analysis from the software evolution models

| **Prediction** |
|---|
| Defects [Khoshgoftaar et al., 1999; Graves et al., 2000; Ohlsson et al., 2001; Arisholm and Briand, 2006; Ratzinger et al., 2007; Shatnawi and Li, 2008; Raja et al., 2009; Krishnan et al., 2011b]; Change [Zhou et al., 2008; Yazdi et al., 2014]; Architecture evolution [Trindade et al., 2017]; Clone Evolution [Antoniol et al., 2001]; Effort [Ramil and Lehman, 2000]; Size [Caprio et al., 2001]; Trends in the evolution of Java systems [Chaikalis and Chatzigeorgiou, 2015] |
| **Characterization** |
| Evolution of software structure [Chen et al., 2008; Zheng et al., 2008; Ferreira et al., 2011; Pan et al., 2011; Bhattacharya et al., 2012; Li et al., 2013; Liu and Ai, 2016]; Evolution of software robustness [Wang et al., 2014] |
| **Descriptive** |
| Explain the evolution of internal properties of the software [Woodside, 1979; Koch, 2005; Turnu et al., 2011; Kirbas et al., 2014]; Detail the evolution development activities over the software life time [Feitelson, 2012]; Compare the evolution trends between traditional and open environments [Capiluppi, 2003] |
| **Simulation** |
| Compare the size evolution between in open and property systems [Smith et al., 2004]; Understand the evolution patterns of size and complexity in open-source software [Smith et al., 2005]; Study the evolution of size, complexity and effort in open-source software [Smith et al., 2006] |

Table 3.12 summarizes the main objects of study of the proposed models. Regarding Prediction, models mostly concentrate on the prediction of defects. There are also prediction models for changes, architecture evolution, clone evolution, effort, size, and trends in the evolution of Java systems. 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 the internal components or type of them grow over time, and how the connection between them changes and increases the system.

The Descriptive models proposed in the literature have aimed to explain the evolution of intrinsic properties of software systems, such as size [Woodside, 1979; Koch, 2005], defects [Kirbas et al., 2014], names of instance variables [Turnu et al., 2011], names of methods [Turnu et al., 2011], calls to methods [Turnu et al., 2011], and sub-classes [Turnu et al., 2011]. Moreover, some other studies have designed descriptive models to detail how development activities evolve and compare the evolution patterns and trends between different environments.

The Simulation models found in the literature are based on the evolution of software systems' internal quality attributes in different contexts. The models were proposed by the same authors and consider the evolution of size [Smith et al., 2004], complexity and size patterns that usually happen in the evolution of open-source software systems [Smith et al., 2005], and size, complexity, and effort [Smith et al., 2006].

**Summary of RQE.4:** 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. Descriptive models have been mainly designed to explain how the software systems' intrinsic properties behave over time. Finally, Simulation models have been built to study and compare the evolution of software systems' internal quality attributes.

## 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's definition is one of the main threats in an SLR since it defines the studies that will be analyzed. If a search string is poorly defined, relevant studies may not be returned. 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 chose the one that returned the most significant number of relevant studies in software evolution. Therefore, we believe that this threat to validity was mitigated, 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 SLR. Not all Software Engineering conferences are indexed in electronic libraries, and therefore, we may miss relevant studies published in such conferences. To mitigate this threat, we chose five electronic libraries of great importance to the academic community. We believe that the choice of these libraries reduced much the chance of missing relevant studies. Besides, we ran a snowballing process during the selection phase of this SLR, i.e., we revisited the reference lists of the selected papers to recover studies that were not retrieved by the SLR.

**Selection Process.** The selection process is a phase in an SLR where the studies retrieved by the search string are analyzed to remove the ones that are not related to

the research topic. The way we carried out this process may be a threat to the validity of an SLR because relevant studies may be discarded. To mitigate this threat, we defined a sequence of five clear and strict steps, where the decision on discard a paper involved analysis of the three authors from this study. Therefore, we believe that the way we defined the steps in this process and the analysis was crucial to mitigate this threat.

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

**Data Extraction.** The author of this thesis project was responsible for analyzing the selected papers and extracting the information that supported the answers to the research questions. One may consider this situation as a threat to the validity of this study. To mitigate this threat, the data gathered from the papers and the decision of discarding a paper were discussed with the advisor and co-advisors from this thesis project.

## 3.5 Final Remarks

This chapter presented an SLR on software evolution. In particular, the study focused on software structure and source code evolution. This SLR had as purpose to compile the knowledge on software evolution produced so far in these topics.

This SLR comprised the analysis of 130 works. We identified five categories of studies on software evolution. They are (i) applicability of Lehman's laws; (ii) application; (iii) evolution of quality attributes; (iv) software structure evolution; and (v) model. This SLR provided the compilation of the main insights and rules defined by the software structure evolution studies. The results of this study lead us to the following main conclusions:

- There is a lack of large datasets with software evolution data.
- Usually, the researchers define their dataset with a low number of systems, and not always they made the dataset available.
- The literature validated the $1^{st}$, $3^{rd}$, and $6^{th}$ Lehman's laws. The $2^{nd}$ law has been partially validated, and the $4^{th}$, $5^{th}$, and $7^{th}$ laws have not been validated.

The 8th law has been little analyzed, and the results about its validation are not conclusive.

- The Lehman's laws have been investigated in five contexts: (i) mobile application; (ii) open-source software; (iii) eclipse third-party plugins; (iv) proprietary software; and (v) C library.
- A set of 33 software evolution applications were identified in this SLR. However, only 16 of them are available for download or on-line use.
- Software visualization is the primary type of tool produced by studies. Java and C++ are the programming languages most used to develop software evolution applications.
- The studies on software evolution considered 14 quality attributes. Size and complexity are the most common ones.
- In total, we found 72 software metrics considered to analyze the evolution of internal quality attributes. Among these metrics, `LOC`, `NOFL`, `CMTL`, and `SIZKB` are the most used to measure size; and `VG`, `CplXLCoh`, and `HVOLU` are often used to evaluate complexity.
- There is a set of 10 software system dimensions that have been objects of study on software evolution. They are the internal structure, bad smell, architectural design, coupling, faults, function side effects, internal quality, project interdependencies, technical debt, and vulnerabilities.
- There are four types of software evolution models proposed in the literature: characterization, descriptive, prediction, and simulation. The prediction model is the prevalent type.
- Many techniques and strategies have been used to build software evolution models. Regression techniques and ARIMA are the most used in prediction models. Regressions techniques are also used in descriptive models. Characterization models usually apply graph-based techniques. Simulation models used qualitative reasoning for its definition.
- We identified 29 metrics used to evaluate the software evolution models. The most used metrics to evaluate prediction and descriptive models are: `R-square`, `Akaike information criterion (AIC)`, `false-positive rate (FPR)`, `mean absolute percentage error (MAPE)`, `mean squared error (MSE)` and `recall`.
- Prediction models mostly concentrate on predicting defects. 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 found only three simulation models, however, they are sub

sequential works done by the same authors. These simulation models are focused on the evolution of internal quality attributes.

Chapter 4 describes the novel method we propose for analysis of software evolution based on time series.

# Chapter 4

# Study Design

In this chapter, we present the method we defined to analyze software evolution data considering coupling – fan-in and fan-out –, inheritance hierarchy – DIT and NOC –, and size – NOA and NOM – metrics. The method comprises two phases: behavior analysis presented in Section 4.1 and trend analysis described in Section 4.2. Our method uses some statistical tests. Therefore, it is essential to mention that in all the statistical analysis, results are considered significant on a 5% level.

## 4.1 Behavior Analysis

In the first phase of our method, we investigate and identify the type of model that best fits the times series and, consequently, explains their behavior pattern. This phase consists of six steps.

**Step 1 - Time series normalization.** The metrics considered in this thesis project are measured at the class level. The dataset has a time series of the metrics for all classes within a system. Therefore, there are many time series in a single system. This step aims to normalize them and extract a global time series for each system. This normalization allows us to model and analyze how the dimensions evolve globally.

As we aim to analyze how the dimensions evolve for the system, we decided to represent the global coupling and size by taking the sum of the values of their metrics regarding the system's classes for each version. Previous works have shown that the values of metrics for coupling and size do not fit a distribution in which the mean value is representative. Such metrics follow a heavy-tailed distribution [Louridas et al., 2008; Filó, 2014; Filó et al., 2015]. For this reason, we chose the sum to represent the global measure of the fan-in, fan-out, NOA, and NOM metrics of a system.

Regarding DIT and NOC, studies in the literature have pointed out that these metrics do not have a high variation in the software systems. DIT usually ranges from 1 to 4, while NOC, also called NSC, often varies between 1 and 3. Moreover, such metrics follow a distribution in which the mean value is representative [Filó, 2014; Filó et al., 2015]. Using the sum for representing the global values of these metrics would not reflect their reality in the systems and could bias our results. Due to this, we decided to describe the general DIT and NOC by taking the arithmetic average of these metrics for each systems' versions. Therefore, we worked with ten global time series, extracted of each system existing in the dataset, for each metric regarding the evolution of the average DIT/NOC.

**Step 2 - Application of linear regression method.** It consists of 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 observations in time series 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 that the time series observations are equally spaced over the total time period. As `COMETS` dataset was built in terms of system versions, we chose the linear regression approach, which it is more flexible and appropriate to our data. Considering that standard linear regression is not efficient with autocorrelated data, which is the case of time series, we also implemented some adjustments. We modeled the metrics of coupling – fan-in and fan-out –, inheritance hierarchy – DIT and NOC –, and size – NOA and NOM – using the following types of model:

**(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 based on all these types of model to evaluate and identify which of them better describes the evolution patterns of the analyzed metrics. It is worth noting that although we used linear regression to model the metrics evolution pattern, our purpose in this part of this thesis project is not to provide a prediction model but to characterize their behavior over the software evolution process.

**Step 3 - Intervention analysis.** Time series may be frequently affected by external factors or events, such as holidays, strikes, policy changes, promotions, weather disasters, and other events. The occurrence of these factors in time series may change the evolutionary behavior of an analyzed phenomenon or even affect its prediction. In the software context, this is not different. A system usually goes through several modifications and refactoring processes over its lifetime. Such processes may change the system's time series pattern over its evolution. To treat this characteristic of time series, we have used intervention analysis [Wei, 2006], a technique that evaluates and measures the effects these external factors cause in the time series. In general, the use of this technique 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 consists of checking the presence of change points that may impact the time series behavior and carrying out an intervention analysis at the systems time series to adjust the model to the new pattern. This process allows us to improve the representation quality of the models.

**Step 4 - 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. If we do not treat this autocorrelation, the estimates of the coefficients and their standard errors may be wrong, and the model will not correctly represent the time series [Bowerman and O'Connell, 1993]. Autocorrelation consists of a serial dependence between their values [Cowpertwait and Metcalfe, 2009]. This step aims to evaluate the models' errors and remove autocorrelation by modeling them via autoregression. Autoregression is a process that uses observations 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 into its respective model.

**Step 5 - Assessment metric.** To assess the models adequacy, we computed their adjusted determination coefficient ($\overline{R}^2$). Both the determination coefficient ($R^2$) and the adjusted determination coefficient ($\overline{R}^2$) are metrics extracted from the analysis based on linear regression. They measure the adjustment of a model to the data so that we may understand to which extent the model explains the variability of analyzed data [Miles, 2014]. However, we decide to choose the ($\overline{R}^2$) instead of $R^2$ because it considers the number of parameters introduced in the model and penalizes the inclusion of less critical parameters.

**Step 6 - Model evaluation.** It aims to compare the best models obtained for the systems time series regarding each type and select the type that better describes

the metrics' behavior. As we are using intervention and autoregression analysis to improve the models' fit, we may get $\overline{R}^2$ values very high and close to each other. Then, we defined an evaluation protocol to compare the values and choose the best model. Our protocol is composed of three stages, and each of them evaluates a different aspect in the model:

1. **Relevance:** it analyzes all generated models and select those with values of $\overline{R}^2$ higher than or equal to 90%.

2. **Coverage:** among the models chosen in the previous stage, coverage selects the ones that cover the most significant number of systems.

3. **Simplicity:** if we pick more than one model in Stage 2, we opt by the simplest model considering the order: (i) linear, (ii) quadratic, (iii) cubic, (iv) logarithmic at Degree 1, (v) logarithmic at Degree 2, and (vi) logarithmic at Degree 3.

## 4.2 Trend Analysis

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

**Step 1 - Data organization.** In the original dataset used in this part of the thesis project, `COMETS` [Couto et al., 2013], when a given class is not present in a given version of the system, its metrics are set to -1 in the data corresponding to that version. In the context of our analysis, -1 values are not representative and may bias the results. Hence, in this step, we remove them from the time series and reorganize the time series by moving the representative values for the classes' first versions.

**Step 2 - Removal of ghost classes.** Over the evolution process of object-oriented software, classes may be included and removed at any moment. These changes may introduce a phenomenon in some classes that we named ghost classes, and we may identify it in the systems time series. This phenomenon consists of breaks in the time-series observations of the classes, dividing them into several small sub-series. Figure 4.1 illustrates this event using the DIT time series representation of a class extracted from `Eclipse JDT Core`. Observing this example, we may see that the class `SourceRefElementInfo` was removed before the $100^{th}$ system's version and reintroduced again after the $150^{th}$. This break in the middle of the time series observation makes the trend analysis unfeasible in these cases. Then, we decided not to analyze

them in this part of our analysis. Therefore, this step aims to identify and remove series with this phenomenon from our analysis. Nevertheless, in this step, no more than 2% of the time series was removed from the application of the trend tests. Hence, ignoring those data will not introduce bias in our analysis.

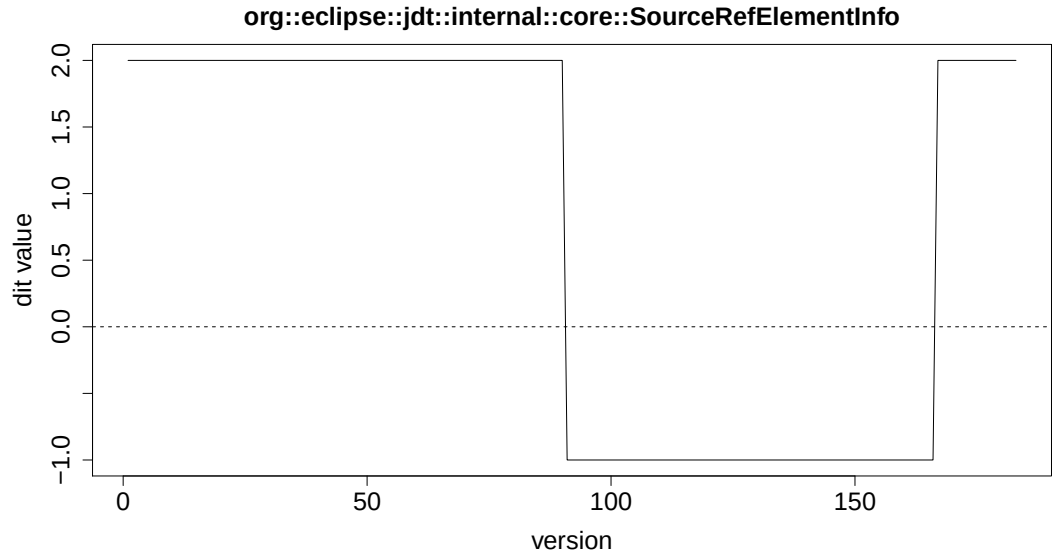


Figure 4.1: Time series of a ghost class.

**Step 3 - Application of trend tests.** It consists of applying the trend tests in the time series to identify whether it has a growth or a decreasing trend. We used three different tests based on hypothesis analysis to define the presence of trends in time series:

**(i)** Mann-Kendall [Kendall, 1975];
**(ii)** Cox-Stuart [Morettin and Toloi, 2006];
**(iii)** Wald-Wolfowitz [Morettin and Toloi, 2006].

We chose these tests because the literature has pointed them as useful and efficient. We consider 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 though we chose relevant and useful tests to identify trends in time series, statistical tests may contain weaknesses and, consequently, be prone to errors. Due to this, we defined an approach based on three statistical trend tests, and defined the following criteria to determine the presence of trend:

*"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 the removal of -1 values from the time series may substantially reduce the number of observations in some of them. Therefore, we decided to analyze and apply the trend tests only in time series with ten or more continuous observations. Times series with less than ten measures were disregarded and classified as having no trend.

**Step 4 - Identification of autocorrelated time series.** Although the three tests applied in Step 3 are useful for detecting trends, the Mann-Kendall test is sensitive to the presence of autocorrelation. According to Hamed and Rao [1998], when we run the original version of the Mann-Kendall test in an autocorrelated time series, it may generate false-positives or false-negatives. This step aims to avoid this problem by analyzing the times series to identify the ones with autocorrelation. To facilitate our analysis, we designed and developed an automatic checking approach for this purpose. This approach analyzes the autocorrelation (ACF) and partial autocorrelation (PACF) plots of the time series. ACF consists of 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 observations. PACF consists of a plot of the partial correlation of the series with its own lagged values, regressed at shorter lags.

Even though the PACF is relevant to identify autocorrelation in time series, it requires attention. A high value at lag 1 (one) of the PACF chart may not indicate autocorrelation, but other time series characteristics, e.g., non-stationarity. A non-stationary time series is the one whose statistical properties change over time, while a stationary time series is the one in which its properties are constant over time [Morettin and Toloi, 2006]. As the non-stationary property of a time series obscures the autocorrelation, it is necessary to apply transformations in the time series to remove non-stationarity. The most common transformation is to take successive differences from the original time series until it becomes stationary. We defined that time series with a value at the lag 1 at the PACF chart between 0.80 and 1 are non-stationary and need be transformed to treat this problem and standardize our approach. Therefore, we defined our automatic checking approach as follows.

Initially, we compute the PACF of the analyzed time series. If the value obtained for that at lag 1 (one) is greater than 0.80, we apply the difference in this time series until it becomes stationary, i.e., its value at lag 1 from the PACF chart be less than 0.80. After removing the non-stationarity of the time series, we plot the ACF and

PACF charts for the analyzed time series. If the value of the lag 1 in the ACF or PACF is greater than the confidence interval generated by the charts, the time series is autocorrelated. However, if the value at the lag 1 is inside the confidence interval in both ACF and PACF charts, the time series are not autocorrelated. Algorithm 1 illustrates the main steps of our automatic checking approach.

**Algorithm 1:** Checking of autocorrelation in time series.

```
   Result: Presence of autocorrelation in the time series,
 1 initialization;
 2 calculate the PACF from the time series;
 3 while lag 1 is greater than 0.80 do
 4 |   apply the difference in the time series;
 5 end
 6 verify lag 1 in the ACF and PACF extracted from the time series;
 7 if lag1 from ACF or PACF is greater than the limit then
 8 |   there is autocorrelation;
 9 else
10 |   there is no autocorrelation;
11 end
```

**Step 5 - Application of the modified Mann-Kendall test.** It aims to apply a modified approach of the Mann-Kendall test to deal with autocorrelated time series. Hamed and Rao [1998] studied the effect of the autocorrelation on the mean and variance of the Mann-Kendall test and derived a theoretical relationship to calculate the variance of the original test for autocorrelated data. Based on these theoretical results, they modified the value of the variance from the original test and proposed a modified approach more suitable and powerful for autocorrelated data. Therefore, we used this approach to analyze the autocorrelated time series.

**Step 6 - Identification of trends.** It consists of applying the criteria defined in Step 3 and identifying the time series with a trend. For the time series in which we identify autocorrelation, we analyzed the p-values resulting from the modified Mann-Kendall test, Cox-Stuart, and Wald-Wolfowitz. For the time series in which we do not identify autocorrelation, we analyze the p-values resulting from the original Mann-Kendall test, Cox-Stuart, and Wald-Wolfowitz.

**Step 7 - Classification of trends.** It consists of evaluating the type of trend in the time series. To do this, we plot the chart of each time series from the systems. We, then, visually analyze the behavior of the time series identified in Step 6. After that, we manually classify them by considering the type of trend as follows.

- **Upward trend:** it is a pattern whose distance between the trend line and x-axis increases over the x-axis.
- **Downward trend:** it is a pattern whose distance between the trend line and x-axis decreases over the x-axis.
- **Undefined trend:** cases that do not follow a clear pattern. We also include here trends of times series whose values of the first and last observations are equal.

## 4.3 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 global time series of the metrics values obtained from the systems and modeling them by applying linear regression techniques. It aims to identify which type of model better describes the evolution of the systems' internal characteristics. Using only the standard linear regression to model autocorrelated 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 3 and 4), to treat these problems and ensure functional model production.

The second phase of our method consists of applying statistical trend tests to the classes' time series 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 dimension in a software system over its evolution process.

# Chapter 5

# Empirical Analysis of Software Evolution

This chapter reports the empirical analyzes we performed to investigate how the object-oriented software systems evolve from the perspective of coupling, size, and inheritance hierarchy. To represent these dimensions, we examined the time series regarding six software metrics. We considered fan-in and fan-out, to represent coupling, NOA (Number of Attributes) and NOM (Number of Methods), to measure size, and DIT (Depth of Inheritance Tree) and NOC (Number of Children), to represent inheritance hierarchy.

Fan-in indicates the number of references made to a given class by other classes, while fan-out reflects the number of calls made by a given type to other classes [Lee et al., 2007b; Sommerville, 2012]. As we showed in Section 2.1, there are many static and dynamic software metrics in the literature that allow measuring coupling in object-oriented software. We decided to use fan-in and fan-out because they consider the method invocations and class attributes as coupling, provide measure 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]. Moreover, 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 using these metrics for representing inheritance hierarchy and size because they are well-known. Consequently, it is available in the literature many tools for supporting their collection and datasets with values regarding these metrics already extracted from the evolution of some software systems. Besides, concerning size metrics, many previous studies used LOC (Lines of Code) to measure this dimension [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., 2007; Koch, 2007; Gonzalez-Barahona et al., 2009; Darcy et al., 2010; Hatton et al., 2017]. Using NOA and NOM to analyze the size evolution in software allows us to provide an evolutionary view of this dimension from the perspective of data and features delegated to the systems.

Therefore, we organize this chapter as follows. Section 5.1 presents the research questions we proposed to investigate throughout these empirical analyzes. Section 5.2 details `COMETS`, the software evolution dataset we used to carry out our empirical studies. Section 5.3 details the evolution of the coupling dimension. Section 5.4 shows the main results regarding the evolution of the inheritance hierarchy. Section 5.5 analyzes the evolution of the size dimension and discusses the main observations obtained from it. Section 5.6 enlists the main potential threats to validity in this part of our research and discusses the decision we took to mitigate them. Section 5.7 concludes this chapter.

## 5.1 Research Questions

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

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

This research question aims to analyze how the software dimensions evolve and identify patterns that better describe their behavior. We have applied the behavior analysis phase of our method, Section 4.1, into the global time series of the software dimensions metrics to model and extract the best evolution pattern that represents them. We carry out this analysis for each dimension studied in this work.

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

This research question aims to analyze how the metrics' relationship impacts the internal dimension over software evolution. For this purpose, we established new measures using the dimension metrics to explain the evolution of some aspects of the software. For instance, we divided fan-in by fan-ou and vice-versa to measure the rate of necessary and unnecessary coupling in the software systems. We also divided NOA by NOM to identify the rate that the proportion of these two metrics evolves.

Regarding the inheritance hierarchy dimension, we have used DIT and NOC metrics to characterize its depth and breadth in the software systems. However, although these metrics express well these aspects of the inheritance hierarchy separately, the relation between them does not provide a direct and well-defined interpretation as those provided by the coupling and size metrics. Therefore, we decided to ignore RQ2 to analyze the inheritance hierarchy and investigate only the RQ1 and RQ3 research questions to this dimension.

***RQ3.*** *What set of classes within the software system affects the dimensions of growth/decrease and how these classes evolve?*

This research question aims to identify the classes responsible for increasing and decreasing the measures of the internal dimensions over the software evolution. Identifying these set of classes, we may quantify the systems' percentage that contributes to the growth and decrease of an aspect in their internal structure. We have applied the trend analysis, reported in Section 4.2 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.

## 5.2 Dataset

We used a public dataset, `COMETS (Code Metrics Time Series)`, composed of time series from 17 well-known software metrics regarding ten open-source Java systems[Couto et al., 2013]. Table 5.1 summarizes the main characteristics of `COMETS`.

There are three other datasets of evolution data: `D'Ambros dataset` [D'Ambros et al., 2010], `Helix` [Vasa et al., 2010], and `Qualitas Corpus` [Tempero et al., 2010]. However, all these datasets have some drawback. For instance, `Qualitas Corpus` does not provide time series from object-oriented metrics, `Helix` does not include time series on coupling metrics, and `D'Ambros dataset` even provides time series on coupling metrics, but it has a smaller number of systems and versions than the `COMETS` dataset. Therefore, we chose `COMETS` because it is the largest and most recent dataset with time series of software metrics. Besides, this dataset comprises all software metrics we apply in this work.

The `COMETS` dataset's time series are sequences of metric values collected from classes of a system over their versions. Each version corresponds to an interval of bi-week, i.e., 14 days. As a system is composed of several classes, the dataset contains many time series. The size of the systems' time series existing in the `COMETS` ranges

Table 5.1: Systems of the COMETS dataset.

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

between a total of 72 and 248 number of versions. The data are available in the COMETS dataset as *CSV* files.

Each *CSV* file corresponds to a specific metric collected from a specific system. The *CSV* has the following format: given a metric $M$ and a system $S$, the lines represent the set of classes from $S$, and the columns represent the versions from $S$. Therefore, each cell *(c, x)* of this file has the value of the metric $M$ extracted for the class $c$ in the version $x$.

It is essential to highlight that since this dataset has metric values of evolving systems, there are cases in which classes existing in a given version are not present in the next version. Therefore, to treat these cases and avoid inconsistencies, the *CSV* creation considered the following restrictions:

**(i)** when a class $c$ does not exist in a version $x$ from a system $S$, the value attributed in the cell *(c, x)* from the respective *CSV* is -1

**(ii)** when class $c$ exists in a version $x$ from a system $S$, the value attributed in the

cell *(c, x)* from the respective *CSV* is the value resulting from the measurement process.

## 5.3 Coupling Evolution

This section presents the main observations we extracted regarding the coupling evolution to answer the specific research questions for this dimension.

### 5.3.1 Coupling Evolution in the System Level

In this section, we investigate **RQ1.** *Which model better describes the evolution pattern of the dimensions in software systems?1*

Coupling is a relevant measure of complexity. Analyzing its behavior over software evolution is essential because the more it increases, the more maintenance effort the system requires. Hence, modeling this dimension may aid developers to understand how it evolves over the software life cycle and allows them to design strategies that control the growth of this characteristic in the systems. We applied regression techniques on the global time series regarding the ten systems from `COMETS` to answer RQ1. We also computed the adjusted determination coefficient ($\overline{R}^2$) to evaluate and compare the resulting adjustment of these models and to identify the one that better describes the pattern of coupling growth.

Before modeling the global time series and evaluating the $\overline{R}^2$ extracted from the generated models, we analyzed fan-in and fan-out metrics' behavior over the software evolution. We plotted the global time series extracted from the software systems as chart lines to analyze if they increase or decrease over time. Figure 5.1 shows the global time series charts regarding fan-in and fan-out.

Analyzing the plots of the global time series regarding fan-in and fan-out, we identify, in all analyzed systems, that both metrics have a growth pattern, and then, tend to increase over the software evolution. Moreover, we observe that fan-out has a growth higher than fan-in in all systems. At the beginning of the `PMD` life cycle, fan-in is slightly higher than fan-out, but this scenario changes quickly in some versions ahead. Fan-out surpasses fan-in and continues to grow at a high frequency throughout the evolution of `PMD`.

After identifying the evolution pattern of fan-in and fan-out, we modeled the global time series of these metrics by using regression techniques and applied our evaluation protocol, Section 4.1, in the $\overline{R}^2$ values obtained from the generated model to detect which type of model better describes and characterizes their global evolution.

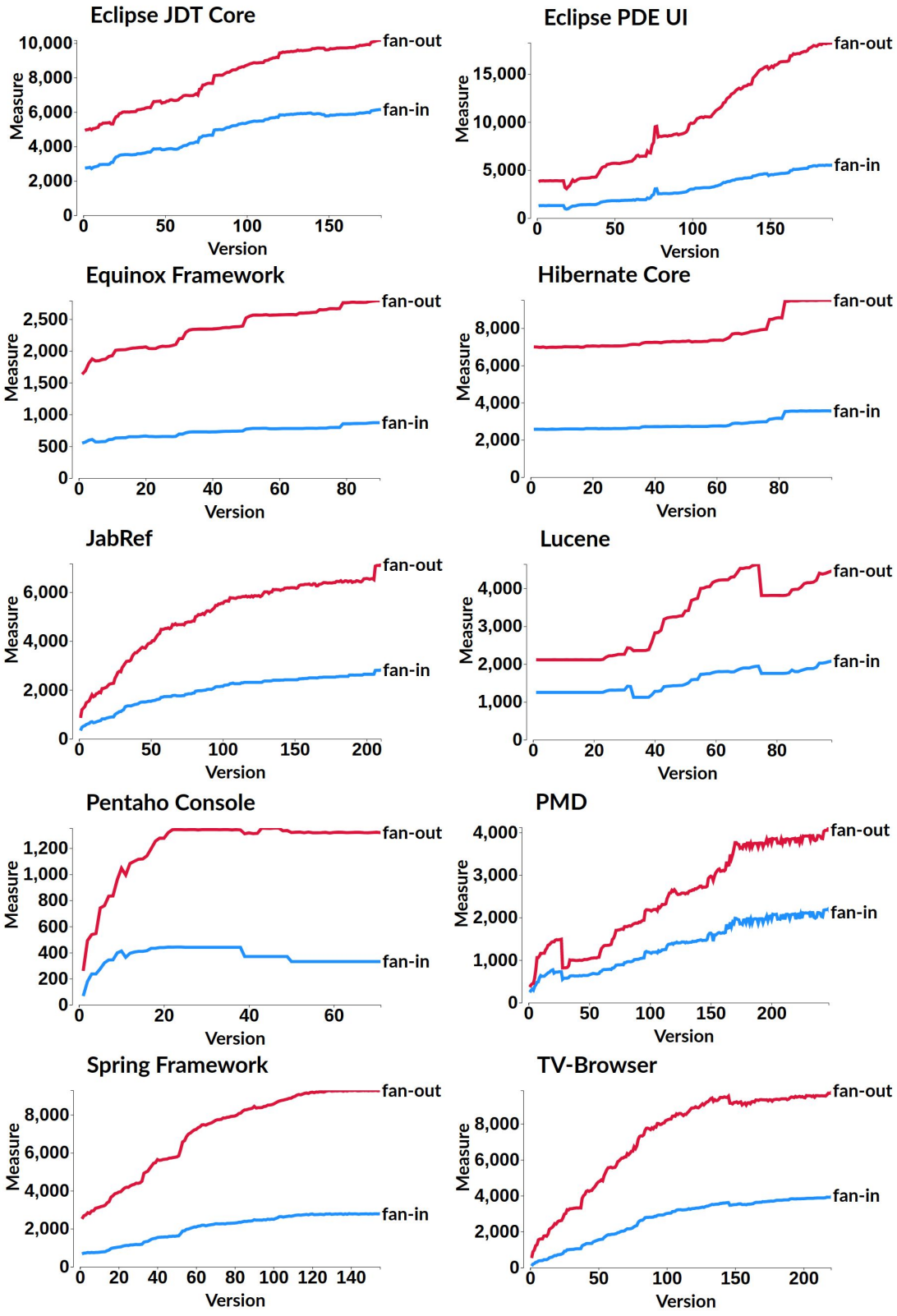


Figure 5.1: Global fan-in/fan-out time series of the analyzed systems.

Tables 5.2 and 5.3 summarize the $\overline{R}^2$ scores computed for the models fitted to fan-in and fan-out metrics, respectively, in each system. 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, in this sequence.

Table 5.2: $\overline{R}^2$ values computed from the fan-in models.

| System | lin. | quad. | cub. | log. 1 | log. 2 | log. 3 |
|---|---|---|---|---|---|---|
| Eclipse JDT Core | 99.84% | 99.84% | 99.84% | 99.82% | 99.82% | 99.77% |
| Eclipse PDE UI | 99.72% | 99.72% | 99.73% | 99.55% | 99.57% | 99.57% |
| Equinox Framework | 98.58% | 98.57% | 98.57% | 98.44% | 98.44% | 98.44% |
| Hibernate Core | 98.55% | 98.58% | - | 98.71% | 98.73% | - |
| JabRef | 99.88% | 99.88% | 99.88% | 99.86% | 99.86% | 99.86% |
| Lucene | 97.65% | 97.63% | 97.61% | 97.28% | - | 97.24% |
| Pentaho Console | 92.64% | 93.66% | 95.11% | 85.70% | 89.40% | 94.94% |
| PMD | 99.25% | 99.24% | - | 99.27% | 99.30% | - |
| Spring Framework | 99.87% | 99.88% | 99.83% | 99.86% | 99.86% | 99.87% |
| TV-Browser | 99.94% | 99.94% | - | 99.89% | 99.66% | 99.87% |

Table 5.3: $\overline{R}^2$ values computed from the fan-out models.

| System | lin. | quad. | cub. | log. 1 | log. 2 | log. 3 |
|---|---|---|---|---|---|---|
| Eclipse JDT Core | 99.85% | 99.85% | 99.83% | 99.83% | - | 99.85% |
| Eclipse PDE UI | 99.84% | 99.84% | 99.74% | 99.75% | 99.76% | 99.85% |
| Equinox Framework | 99.39% | 99.42% | 99.31% | 99.38% | 99.37% | 99.41% |
| Hibernate Core | 98.63% | 98.68% | 98.78% | 98.79% | - | - |
| JabRef | 99.84% | 99.84% | 99.68% | 99.70% | 99.78% | 99.85% |
| Lucene | 98.67% | 98.69% | 99.05% | 99.09% | 99.09% | 98.68% |
| Pentaho Console | 98.27% | 98.25% | 97.52% | 97.50% | 97.58% | 98.28% |
| PMD | 99.55% | 99.55% | 99.03% | 99.04% | 99.06% | 99.57% |
| Spring Framework | 99.92% | 99.92% | 99.89% | 99.89% | 99.89% | 99.92% |
| TV-Browser | 99.89% | 99.90% | 99.61% | 99.75% | 99.86% | 99.90% |

To improve our discussion of the obtained results, we adopted a color scheme highlighting the models selected in each stage of our protocol. Green color indicates the models selected at Stage 1. Yellow shows the ones chosen in Stage 2, and Red specifies the only model type selected at Stage 3, which is the one that better characterizes the pattern evolution of the metrics by following the parsimonious criteria.

Observing the obtained results for fan-in in Table 5.2, we identify that most models produced good $\overline{R}^2$ scores, generally higher than 90%. We find some exceptions, such as models not generated and values less than 90%. We did not select these

exceptions in Stage 1 of our protocol, and consequently, did not highlight them with the green color. For the initially selected models, we observed that two types of models, linear and quadratic, attended our Stage 2 since both of them produced good fits for all analyzed systems. However, although both linear and quadratic models are efficient in describing the evolution of fan-in, applying the parsimonious criteria, we conclude that the linear model is the one that better explains the growth evolution pattern of fan-in since it has attended all aspects of our evaluation protocol.

When analyzing the results of the fan-out in Table 5.3, we also observe that most of the returned models produced good $\overline{R}^2$ values. We had only three cases not selected at Stage 1 since our method was not able to find models that represent them. Due to this, we did not highlight these exceptions in green. Among the selected models at Stage 1, linear, quadratic, cubic, and logarithmic at degree 1, attended the coverage criteria. All of them produced models for all analyzed systems with $\overline{R}^2$ values higher than 90%. However, by the simplicity criteria in Stage 3, we conclude that the linear model is the one that better describes the pattern evolution of fan-out.

**Summary of RQ1 - Coupling.** In response to RQ1, we identified that fan-in and fan-out have a growth pattern that may be better explained by a linear model. However, although the same model better explains fan-in and fan-out growth patterns, we identified a difference between these two metrics since the fan-out values are much higher than those of fan-in values.

### 5.3.2 Evolution of Fan-in/Fan-out Relation

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

Berard [1993] categorizes coupling at the package level in 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. [2007b], a high fan-in may represents a good object design and high reuse, since classes at the same package are reused together. In contrast, a high fan-out is not desirable in software because it is an indication of 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 as service provider. On the other hand, the *unnecessary coupling* is the

ratio of fan-out by fan-in. A high unnecessary coupling of a class indicates that the main role of the class is as 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 version of the systems. Figure 5.2 summarizes the behavior of these two types of coupling. The acronyms "UNCP" and "NCP" in this figure refer to "unnecessary coupling" and "necessary coupling", respectively.

Analyzing Figure 5.2, we observe that the unnecessary coupling is much higher than the necessary coupling in all systems. This behavior is because fan-out grows quickly and fan-in grows slowly, as shown Figure 5.1. This finding shows that using service from other classes is the prevalent role of the classes within a system. Although this conclusion is not surprising, this analysis shows how prevalent using services is and how it evolves. In 50% of the systems, both necessary and unnecessary couplings do not suffer relevant changes over time. This fact happens with `Eclipse JDT`, `Eclipse PDE UI`, `Equinox`, `Hibernate`, and `JabRef`. In `Pentaho`, the necessary coupling slightly decreased; however, the unnecessary coupling had a different behavior: it decreased over the first releases, continuously increased for many releases, and is stable in the last releases. This fact means that, in the beginning, the new classes inserted in the system were more service users than providers. After, for a long time, the classes inserted in the system were increasingly more service providers, or the other classes turned to provide more services. In the last versions, the inclusion of new classes kept unnecessary coupling stable. In 20% of the systems, `Lucene`, and `PMD`, the unnecessary coupling slightly increased. There was decreasing in the unnecessary coupling only in 20% of the systems, `Springer` and `TV-Browser`. However, in these systems, the necessary coupling remained stable.

**Summary of RQ2 - Coupling.** The unnecessary coupling is much higher than the necessary coupling since the first releases of a system. This means that the rate of classes behaving as service users is higher than the service providers. In most cases, the evolution of the system does not change the relation between fan-in and fan-out in the systems.

### 5.3.3 Coupling Growth/Decrease Analysis

This section answers **RQ3.** *What set of classes within the software system affects the dimensions of growth/decrease and how these classes evolve?*

To answer RQ3, we identified the classes responsible for increasing and decreasing the coupling in the systems. Then, we carried out a trend analysis in the time series

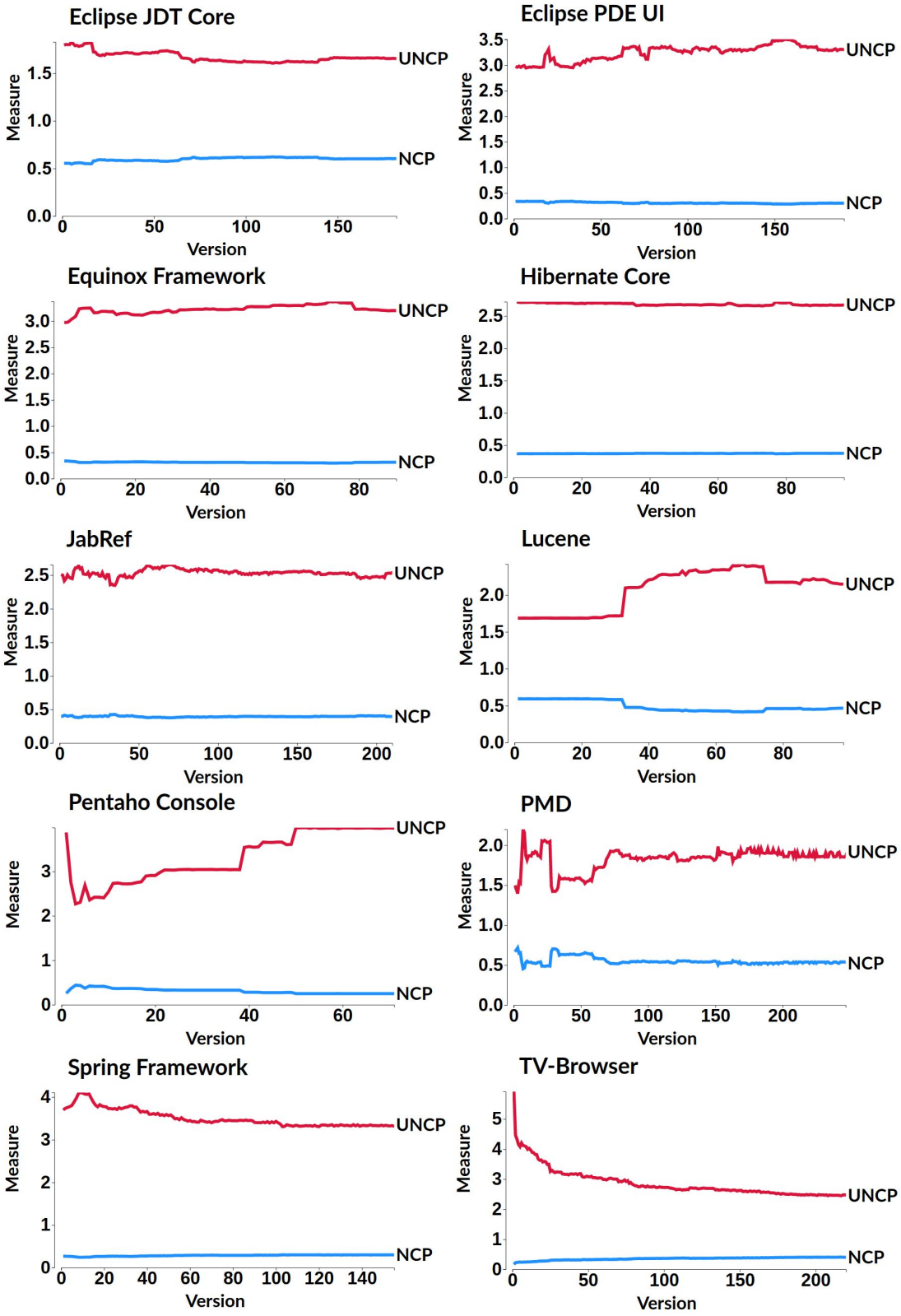


Figure 5.2: Evolution of unnecessary and necessary coupling.

of the systems' classes and computed the percentage of classes whose fan-in and fan-out have increased/decreased over time. Figure 5.3 presents the distribution of the percentage of classes with fan-in growth, fan-in decrease, fan-out growth, and fan-out decrease. In the chart, each box was generated with ten values, i.e., one percentage value per system. Table 5.4 provides a descriptive analysis in terms of percentiles of the boxplot in Figure 5.3. In the next sections, we present the analysis the results regarding coupling growth and decrease.

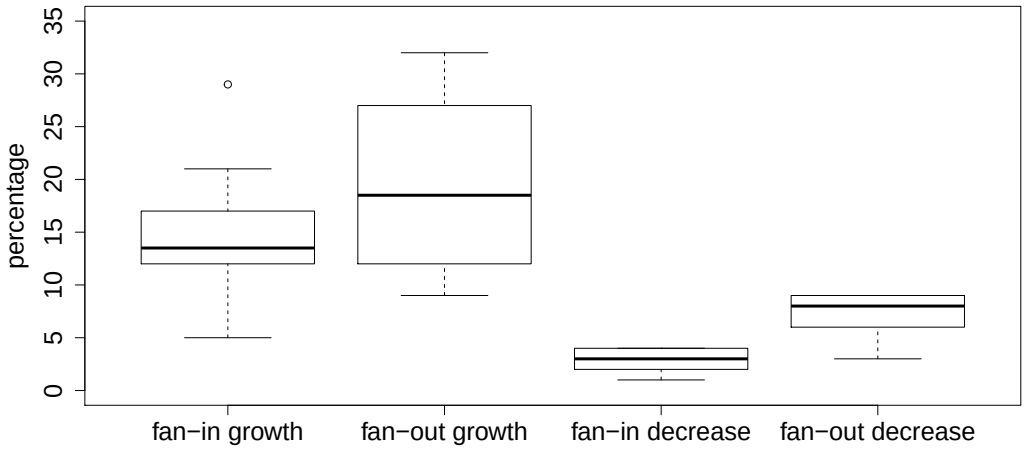


Figure 5.3: Percentage distribution of classes within the systems that impact on coupling growth/decrease.

Table 5.4: Descriptive Analysis of the Percentage Distribution of Classes in the Systems that Impact on Coupling Growth/Decrease.

| Event | 0% | 25% | 50% | 75% | 100% |
|---|---|---|---|---|---|
| Fan-In Growth | 5.00 | 12.00 | 13.50 | 16.75 | 29.00 |
| Fan-Out Growth | 9.00 | 12.50 | 18.50 | 25.25 | 32.00 |
| Fan-In Decrease | 1.00 | 2.25 | 3.00 | 4.00 | 4.00 |
| Fan-Out Decrease | 3.00 | 6.25 | 8.00 | 8.75 | 9.00 |

#### 5.3.3.1 Classes Responsible for Coupling Growth

Figure 5.3 and Table 5.4 show that the median percentages of classes in the systems responsible for "fan-in growth" and "fan-out growth" are 13.50% and 18.50%, respectively. The maximum percentages we have found were 29% and 32% for fan-in and

fan-out, respectively. Nevertheless, it is essential to highlight that 29% is an outlier in Figure 5.3 because only one of the systems presents this value.

The results for fan-in and fan-out couplings show that although they have a growth pattern that is better described by a linear model, their growth is directly influenced by a small group of classes, corresponding to no more than 35% of the classes within a system. We also analyzed how these classes over the versions of the systems. We identified that in 50% of the systems, more than 25% of the classes that contribute to fan-in and fan-out growth are introduced in the first version of the system and remain over the system evolution.

#### 5.3.3.2 Classes Responsible for Coupling Decrease

Observing the results shown in Figure 5.3 and Table 5.4, we note that the median of the percentages of classes responsible for "fan-in decrease" and "fan-out decrease" is 3% and 8%, respectively. The maximum percentages are 4% and 9% for fan-in and fan-out, respectively. This result shows that a low percentage of classes, no more than 10%, impact the coupling decrease. Besides, there is a significant discrepancy between the results obtained for the growth and decrease of coupling. We also analyze the distribution of the classes that impact on coupling decrease. Just as in coupling growth analysis, we identified that in 50% of the systems, legacy classes represent more than 25% of the classes that impact on coupling decrease.

#### 5.3.3.3 Growth versus Decrease from the System Perspective

We analyzed the percentage of classes that have the following behaviors: (i) fan-in and fan-out grow, (ii) fan-in and fan-out decrease, (iii) fan-in grows and fan-out decreases, and (iv) fan-in decreases and fan-out grows. We analyze these cases considering two different perspectives: system and trend classes. Firstly, we analyzed these behaviors from the system perspective as a whole. Table 5.5 presents the percentages obtained in these cases. We computed the percentages in Table 5.5 by dividing the number of intersections by the total classes from the software systems.

The behavior of Case *(i)* has the highest chance to occur since it has the maximum percentages, 18%, and 14%, respectively, and presents a higher percentage than the other cases in all systems. Even though, analyzing the distribution of the percentages in the Case *(i)*, we note that both 18% and 14% are outliers, and the median and maximum values are 5.5% and 8%, respectively, by disregarding these two outliers.

Analyzing the percentages of the four behavior aforementioned, considering only the trend classes, we found the results shown in Table 5.6. We computed the table's

Table 5.5: Intersection of the trend results for fan-in and fan-out from the system perspective.

| System | i | ii | iii | iv |
|---|---|---|---|---|
| Eclipse JDT Core | 18% | 1% | 3% | 1% |
| Eclipse PDE UI | 7% | 1% | 1% | 1% |
| Equinox Framework | 7% | 0% | 1% | 2% |
| Hibernate Core | 4% | 0% | 1% | 0% |
| JabRef | 8% | 0% | 1% | 1% |
| Lucene | 4% | 0% | 1% | 1% |
| Pentaho Console | 1% | 0% | 2% | 0% |
| PMD | 4% | 1% | 1% | 1% |
| Spring Framework | 7% | 1% | 2% | 1% |
| TV-Browser | 14% | 1% | 1% | 1% |

values by dividing the number of intersection by the number of classes that presented any type of trend for fan-in or fan-out.

Table 5.6: Intersection of the trend results for fan-in and fan-out from the trend class perspective.

| System | i | ii | iii | iv |
|---|---|---|---|---|
| Eclipse JDT Core | 35% | 2% | 6% | 2% |
| Eclipse PDE UI | 21% | 2% | 4% | 2% |
| Equinox Framework | 20% | 1% | 3% | 5% |
| Hibernate Core | 15% | 1% | 2% | 2% |
| JabRef | 19% | 1% | 3% | 3% |
| Lucene | 15% | 1% | 3% | 4% |
| Pentaho Console | 4% | 0% | 7% | 0% |
| PMD | 15% | 2% | 2% | 2% |
| Spring Framework | 15% | 2% | 4% | 3% |
| TV-Browser | 29% | 2% | 3% | 3% |

Considering the percentages from the trend class perspective in Table 5.6, we observed that they increased compared to the percentage from the system perspective. However, the values are still not too high. The Case *(i)*, as in Table 5.5, is the one whose percentages stand out concerning the other cases. While Cases *(ii)*, *(iii)* and *(iv)* have percentages that do not exceed 10%, the percentages of Case *(i)* are all higher 10%, except in `Pentaho Console`. Therefore, in general, there is a low percentage of classes that follow the patterns analyzed for the cases in both system and trend class perspectives. Although Case *(i)* has stood out in both system and trend class perspectives, such results suggest evidence that most of the classes that directly impact the coupling evolution do not tend to follow a combined pattern in

terms of growth and decrease of fan-in and fan-out.

**Summary of RQ3 - Coupling.** Coupling evolution is affected by a small group of classes in a system. There is a strong influence of legacy classes on the coupling growth and decrease in 50% of the analyzed systems. We consider a legacy class in this analysis as the one introduced in the first version of a system, and not removed during its evolution. Moreover, the growth/decrease of fan-in and fan-out of the classes that directly impact the coupling evolution do not evolve in an associated way during the software evolution.

## 5.4 Inheritance Hierarchy Evolution

This section presents the results we found regarding the inheritance hierarchy evolution by answering the specific research questions for this dimension. Considering inheritance, RQ2 does not apply because there is no intuitive relation between the DIT and the NOC of a class.

### 5.4.1 Inheritance Evolution in the System Level

This section presents the results of the investigation of **RQ1.** *Which model better describes the evolution pattern of the dimensions in software systems?*

The analysis presented in this section aims to investigate how the inheritance hierarchy evolves over the software evolution and identify the pattern that better describes this behavior.

We applied our method based on regression techniques to the global DIT and NOC time series from `COMETS'` systems. For each release of a system, the DIT's global value is given by the average of DIT classes. The same occurs with NOC. After modeling the evolution of the metrics, we computed the adjusted determination coefficient ($\overline{R}^2$).

Before analyzing the adjusted determination coefficient ($\overline{R}^2$) of the generated models, we observed these metrics' behavior. We plotted the global metrics time series from the systems as line charts. In these charts, we evaluated if the metrics increase or decrease over time. Figure 5.4 shows the global time series charts regarding DIT and NOC.

Given the DIT evolution shown in Figure 5.4, we observed that, for five systems, the global average of DIT increases over the software evolution. More specifically, the

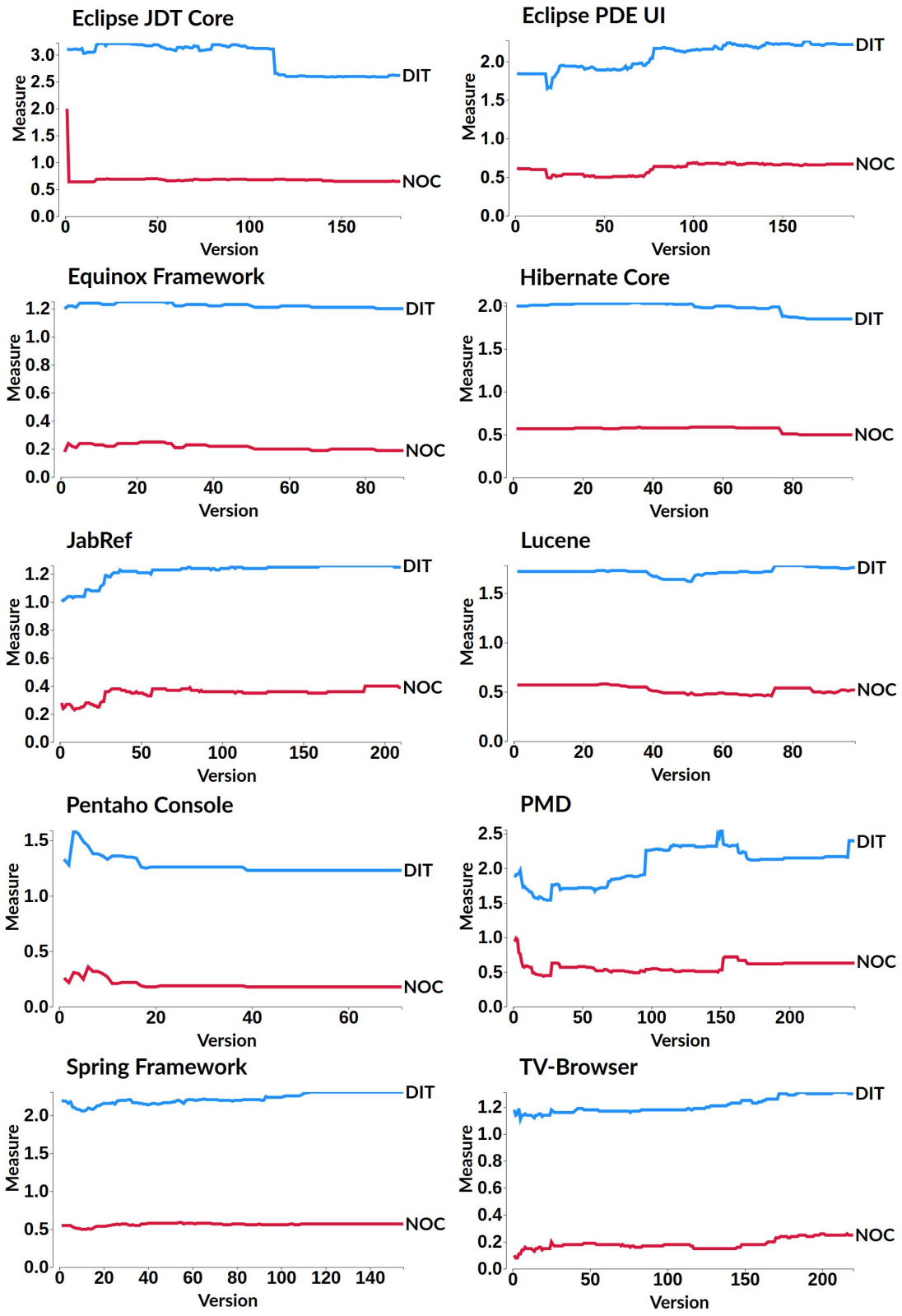


Figure 5.4: Global DIT/NOC time series of the analyzed systems.

systems where this phenomenon happens are `Eclipse PDE UI`, `JabRef`, `PMD`, `Spring Framework`, and `TV-Browser`. `Eclipse JDT Core` and `Hibernate Core` DIT decrease slowly and smoothly. `Pentaho Console` presents an increase of DIT at the beginning, then DIT decreases along some releases and, then, such decreasing became slow and smooth over time. We noted a high decrease of the global DIT in `Eclipse JDT Core` between the $100^{th}$ and $120^{th}$ versions, which might be the result of a refactoring or restructuring process in this system. However, the global behavior of DIT in this software, before and after this event, decreases very smoothly and is almost stable. Moreover, we observed that the global average of DIT in the `Equinox Framework` and `Lucene's` life cycle is practically constant, although there were some smooth variations in the time series of these systems. Then, we can infer that the global average of DIT tends to increase slightly over the software evolution in the majority of the systems.

Concerning the NOC evolution, we analyzed the charts in Figure 5.4 and identified that, in 60% of the systems, NOC decreases over time and, in some cases, such decreasing is very small. The systems presenting this pattern are `Eclipse JDT Core`, `Equinox Framework`, `Hibernate Core`, `Lucene`, `Pentaho Console`, and `PMD`. In `Eclipse JDT Core`, the series starts with a high value and has a drop shortly after that. Then, the global average of NOC follows a decreasing pattern very smooth, remaining almost stable over the whole system life cycle. On the other hand, in some systems, the global NOC has a growth behavior over their life cycle. The systems that presented this pattern are `Eclipse PDE UI`, `JabRef`, and `TV-Browser`. In `Spring Framework`, although the global NOC time series has smooth variations over time, it follows a stable pattern and remains practically constant over this system's life cycle.

Therefore, we conclude by this analysis that the global average of NOC decreases over the software evolution, tending to achieve zero. This result suggests that classes do not have many children in the software context, and they tend to reduce this number.

After identifying the evolution pattern of DIT and NOC, we modeled the global time series of these metrics. We applied our evaluation protocol, Step 7 (Section 4.1), in the $\overline{R}^2$ extracted from the generated models to find the best way to represent and characterize their global evolution pattern. Tables 5.7 and 5.8 show the results of DIT and NOC, respectively. As described before, the labels "lin.", "quad.", "cub.", "log. 1", "log. 2" and "log. 3" indicate, respectively, the $\overline{R}^2$ scores extracted for linear, quadratic, cubic, logarithmic at degree 1, logarithmic at degree 2, and logarithmic at degree 3 models.

For a better comprehension of our discussion, we used the same color scheme of Section 5.3.1 to highlight the models selected at each stage of our evaluation protocol. Analyzing Table 5.7, we observe that most of the produced models has an $\overline{R}^2$ score

Table 5.7: $\overline{R}^2$ values computed from the DIT models.

| System | lin. | quad. | cub. | log. 1 | log. 2 | log. 3 |
|---|---|---|---|---|---|---|
| Eclipse JDT Core | 98.00% | 97.41% | 98.28% | 98.07% | 97.48% | 98.35% |
| Eclipse PDE UI | 97.81% | 97.87% | 97.83% | 97.45% | 97.53% | 97.52% |
| Equinox Framework | 86.89% | 85.01% | 86.49% | 86.95% | 85.10% | 86.54% |
| Hibernate Core | 96.54% | 96.67% | 96.58% | 96.58% | 96.71% | 96.59% |
| JabRef | 98.69% | 98.69% | 98.79% | 98.68% | 98.67% | 98.64% |
| Lucene | 92.23% | 92.21% | 89.52% | 92.13% | 92.11% | 89.42% |
| Pentaho Console | 75.49% | 83.43% | 90.92% | 77.50% | 85.05% | 91.62% |
| PMD | 97.61% | 97.64% | 96.52% | 97.79% | 97.82% | 96.84% |
| Spring Framework | 97.21% | 97.04% | 97.08% | 97.09% | 96.98% | - |
| TV-Browser | 98.24% | 98.09% | 97.94% | 98.11% | 97.97% | 97.81% |

Table 5.8: $\overline{R}^2$ values computed from the NOC models.

| System | lin. | quad. | cub. | log. 1 | log. 2 | log. 3 |
|---|---|---|---|---|---|---|
| Eclipse JDT Core | 88.99% | 91.61% | 91.99% | 89.08% | 91.72% | 92.11% |
| Eclipse PDE UI | 97.13% | 96.69% | 97.16% | 96.92% | 96.46% | 96.95% |
| Equinox Framework | 86.05% | 67.85% | 85.70% | 86.68% | 68.08% | 86.28% |
| Hibernate Core | 94.67% | 94.93% | - | 94.80% | 95.12% | - |
| JabRef | 94.81% | 94.79% | 95.05% | 94.71% | 94.70% | 94.99% |
| Lucene | 96.07% | 92.03% | - | 95.86% | 91.43% | - |
| Pentaho Console | 76.80% | 76.83% | 78.40% | 79.45% | 81.35% | 82.52% |
| PMD | 93.83% | 93.83% | 93.43% | 93.86% | 93.84% | 93.83% |
| Spring Framework | 90.57% | 90.67% | 91.48% | 90.76% | 90.91% | 91.63% |
| TV-Browser | 96.57% | 96.74% | 96.74% | 95.08% | 94.79% | 95.26% |

higher than 90%. In the case of `Equinox Framework`, there is no model with $\overline{R}^2$ higher than or equal to 90%. Figure 5.4 shows that the evolution of `Equinox Framework` is very stable with minimal variations in the DIT time series. This chart shows that the `Equinox Framework` contains a very smooth trend in the global DIT time series, and therefore, the models were not able to capture this trend and generate suitable adjustments to this particular time series. For the initially selected models, we identify that linear, quadratic, cubic, logarithmic at degree 1, and logarithmic at degree 2 attend Stage 2 of our evaluation protocol. However, by applying the simplicity criteria, we conclude that the linear model is the one that better explains the growth evolution pattern of the DIT global average since it attends all aspects of our evaluation protocol.

Observing Table 5.8, we realize that most of the generated models have $\overline{R}^2$ values higher than 90%. However, in the cases of `Equinox Framework` and `Pentaho Console`, there is no model with $\overline{R}^2$ higher than or equal to 90% for the global NOC. Analyzing the chart of the NOC global time series of these systems in Figure 5.4, we

observe many occurrences of interventions and change points in the `Pentaho Console` time series, what avoid to define a function to model it with a satisfactory $\overline{R}^2$ value. Concerning `Equinox Framework`, the trend in the global NOC time series is very smooth, almost constant, then the models were not able to capture this tendency to generate proper adjustments for this system. Two models, quadratic and logarithmic at degree 2, attend Stage 2 of our evaluation protocol. By applying the simplicity criteria, we conclude that the quadratic model better describes the decrease pattern of NOC. Besides, considering that the global average of NOC does not decrease fast, and there are several fluctuations between the system's versions, the quadratic function is the one that best adjusts to them.

**Summary of RQ1 - Inheritance.** The results show that the inheritance hierarchy slightly grows in depth and decreases in breadth over the evolution. We conclude that a linear model better explains the DIT growth pattern, and a quadratic-order model better describes the decrease pattern of NOC.

### 5.4.2 Inheritance Growth/Decrease Analysis

This section describes the analysis of the inheritance hierarchy to answer RQ3. *What set of classes within the software system affects the dimensions of growth/decrease and how these classes evolve?*

To answer this research question, we initially identified the classes existing in the analyzed systems that are responsible for increasing and decreasing the depth or the breadth of their inheritance hierarchy. Then, we carried out the trend analysis considering the time series regarding the systems' classes and extracted the percentage of classes that had growth/decrease in DIT and the classes with growth/decrease in NOC. Figure 5.5 presents the distribution of percentages obtained by DIT growth, DIT decrease, NOC growth, and NOC decrease. Table 5.9 details Figure 5.5 by presenting a descriptive analysis of the data. Following, we discuss the results regarding the growth and the decrease of the inheritance hierarchy separately.

#### 5.4.2.1 Classes Responsible for Inheritance Hierarchy Growth

Figure 5.5 and Table 5.9 show that a tiny percentage of classes in the systems contribute to the growth of DIT and NOC. The median percentages of classes responsible for "DIT growth" and "NOC growth" are 3.00% and 2.00%, respectively. The maximum percentages are 21.00% and 8.00%, respectively. Therefore, the results indicate that a

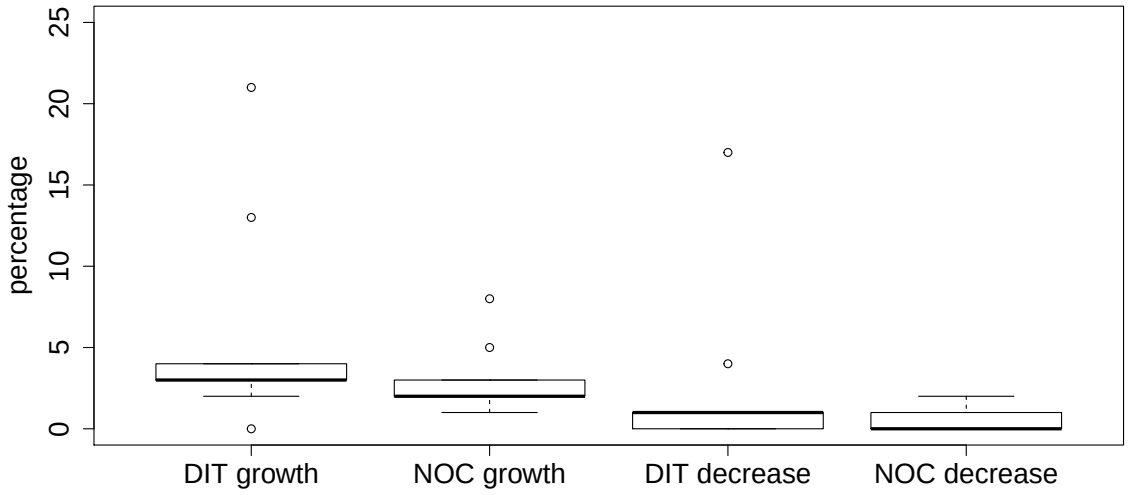


Figure 5.5: Distribution of classes that affects the inheritance hierarchy growth/decrease.

Table 5.9: Descriptive analysis of the distribution of classes that affects the inheritance hierarchy growth/decrease.

| Event/Percentile | 0% | 25% | 50% | 75% | 100% |
|---|---|---|---|---|---|
| DIT Growth | 0.00 | 3.00 | 3.00 | 4.00 | 21.00 |
| NOC Growth | 1.00 | 2.00 | 2.00 | 2.75 | 8.00 |
| DIT Decrease | 0.00 | 0.00 | 1.00 | 1.00 | 17.00 |
| NOC Decrease | 0.00 | 0.00 | 0.00 | 1.00 | 2.00 |

small group of classes in a system affects the inheritance tree's growth directly, in depth and breadth. Regarding DIT, which tends to increase over time, this group represents no more than 21.00% of the system's classes. For NOC, which tends to decrease over time, this group is even less and represents no more than 8.00% of the system's classes. Moreover, these values are outliers, as shown in the boxplot of Figure 5.5.

#### 5.4.2.2 Classes Responsible for Inheritance Hierarchy Decrease

Analyzing the decrease of DIT and NOC in Figure 5.5 and Table 5.9, we note that a small percentage of classes within a system directly influence these metrics to decrease over time. The median of the portion of classes within the systems with a decreasing DIT is 1.00%. For NOC, 50% of the analyzed systems do not present classes with decreasing trends. Besides, the maximum percentages for "DIT decrease" and "NOC decrease" are 17.00% and 2.00%, respectively.

These results show that just a tiny group of classes directly contributes to decreasing the depth and breadth of their inheritance tree. As we identified that DIT tends to increase over time, we expected that the percentage of classes with decreasing DIT would be less than the percentage of classes with increasing DIT patterns. In contrast, NOC has a decreasing trend, but the percentage of classes with a decreasing NOC in a system is less than that with a growing NOC. The possible explanation is: (1) if the number of classes in the system has not grown, decreasing in NOC values of the classes within the system is more intense than the increasing; (2) if the number of classes in the system has grown, most of the new classes will not have children. As usual, the number of classes increases; the second hypothesis is more likely.

#### 5.4.2.3 Growth versus Decrease

In this part of our study, we analyzed the intersection of trend results about DIT and NOC together to identify the percentage of classes with the following behaviors: (i) DIT and NOC growth, (ii) DIT and NOC decrease; (iii) DIT growth and NOC decrease; (iv) DIT decrease and NOC growth. As we did for fan-in/fan-out, we investigated these behaviors from the perspective of system and trend classes. Table 5.10 summarizes the results obtained for these cases considering the system perspective. We computed the percentages reported into it by dividing the number of intersections by the total number of classes existing in the systems.

Table 5.10: Intersection of the trend results for DIT and NOC from the system perspective.

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

Observing Table 5.10, we do not identify high percentages for these cases. In cases *(ii)* and *(iii)*, the percentages of classes are too small that they did not even represent 1% of the classes within the systems. In cases *(i)* and *(iv)*, although some systems had values greater than 0, most of them have minimal percentage of classes.

When analyzing the intersection percentages from the perspective of the trend classes, we found the results reported in Table 5.11. To compute the values in this table, we divided the number of intersections by the number of classes that presented any trend for DIT or NOC.

Table 5.11: Intersection of the trend results for DIT and NOC from the perspective of trend classes.

| System | i | ii | iii | iv |
|---|---|---|---|---|
| Eclipse JDT Core | 3% | 1% | 1% | 15% |
| Eclipse PDE UI | 4% | 0% | 0% | 5% |
| Equinox Framework | 0% | 0% | 0% | 0% |
| Hibernate Core | 0% | 0% | 0% | 3% |
| JabRef | 0% | 0% | 0% | 0% |
| Lucene | 4% | 0% | 0% | 0% |
| Pentaho Console | 0% | 0% | 0% | 0% |
| PMD | 1% | 0% | 0% | 0% |
| Spring Framework | 6% | 0% | 1% | 0% |
| TV-Browser | 0% | 0% | 1% | 0% |

Analyzing Table 5.11, we observe that, as in Table 5.10, the percentages are too small. Despite some exceptions, such as the Case *(iv)* in `Eclipse JDT CORE` and Case *(i)* in `Spring Framework`, the other cases did not present relevant percentages. These results brings evidences that DIT and NOC of a class evolve separately over the software life cycle and do not tend to follow a combined pattern or establish any relation over the software evolution.

**Summary of RQ3 - Inheritance:** The evolution of the depth and the breadth of inheritance trees are directly affected by a small set of classes. Besides, we also identified that the evolution of DIT and NOC of a class evolve in nonrelated patterns.

## 5.5 Size Evolution

This section presents the analysis of size evolution by answering the specific research questions for this dimension.

### 5.5.1 Size Evolution in the System Level

This section answers **RQ1.** *Which model better describes the evolution pattern of the dimensions in software systems?*

We evaluate the dimension size in the class level, in terms of number of attributes (NOA) and number of methods (NOM). This analysis aims to identify the best type of model that describes class size's evolution pattern, given by the metrics NOA and NOM. For each release of a system, the global value of NOA is given by the sum of the NOA classes. The same occurs with NOM. Like the analysis described in Sections 5.3.1 and 5.4.1, we applied regression techniques on the global NOA and NOM time series.

However, before modeling the global NOA and NOM time series, we analyzed these metrics' behavior over the software evolution. Therefore, we plotted the global time series extracted from the analyzed software systems as chart lines. Figure 5.6 shows the global time series charts regarding NOA and NOM.

Analyzing Figure 5.6, we realized that, in all the analyzed systems, NOA and NOM have a growth behavior, i.e., the systems tend to increase their size in terms of attributes and methods over their life cycle. Although NOA and NOM grow over the software evolution, when comparing their global time series, we observed that NOM values are hugely higher than NOA in all analyzed systems, considering the absolute values in the time series. The exception is `JabRef` because NOA starts higher than NOM and follows this configuration up to the $17^{th}$ release. Then, NOM exceeds NOA and continues to grow faster than NOA over the `JabRef` evolution.

After observing the evolution pattern of NOA and NOM, we modeled the global time series of these metrics and assessed the generated models with our evaluation protocol (Section 4.1). Tables 5.12 and 5.13 summarize the resulting $\overline{R}^2$ of NOA and NOM modeling, respectively. As in Sections 5.3.1 and 5.4.1, the "lin.", "quad.", "cub.", "log. 1", "log. 2", and "log. 3" columns indicate the $\overline{R}^2$ scores extracted for linear, quadratic, cubic, logarithmic at degree 1, logarithmic at degree 2, and logarithmic at degree 3 models, respectively.

Table 5.12: $\overline{R}^2$ values computed from the NOA models.

| System | lin. | quad. | cub. | log. 1 | log. 2 | log. 3 |
|---|---|---|---|---|---|---|
| Eclipse JDT Core | 99.82% | 99.82% | 99.83% | 99.77% | 99.79% | 99.79% |
| Eclipse PDE UI | 99.73% | 99.73% | 99.73% | 99.61% | 99.56% | 99.51% |
| Equinox Framework | 98.25% | 98.24% | 98.22% | 97.68% | 97.86% | 97.84% |
| Hibernate Core | 98.75% | 98.79% | - | 98.86% | 98.88% | - |
| JabRef | 99.81% | 99.81% | 99.82% | 99.70% | 99.71% | 99.77% |
| Lucene | 99.30% | 98.99% | 99.29% | 99.44% | 99.03% | 99.45% |
| Pentaho Console | 98.11% | 98.09% | 98.25% | 97.52% | 97.49% | 97.90% |
| PMD | 99.51% | 99.51% | 99.53% | 98.98% | 98.92% | 98.62% |
| Spring Framework | 99.93% | 99.94% | 99.92% | 99.91% | 99.91% | 99.91% |
| TV-Browser | 99.90% | 99.90% | 99.90% | 99.42% | 99.58% | 99.85% |

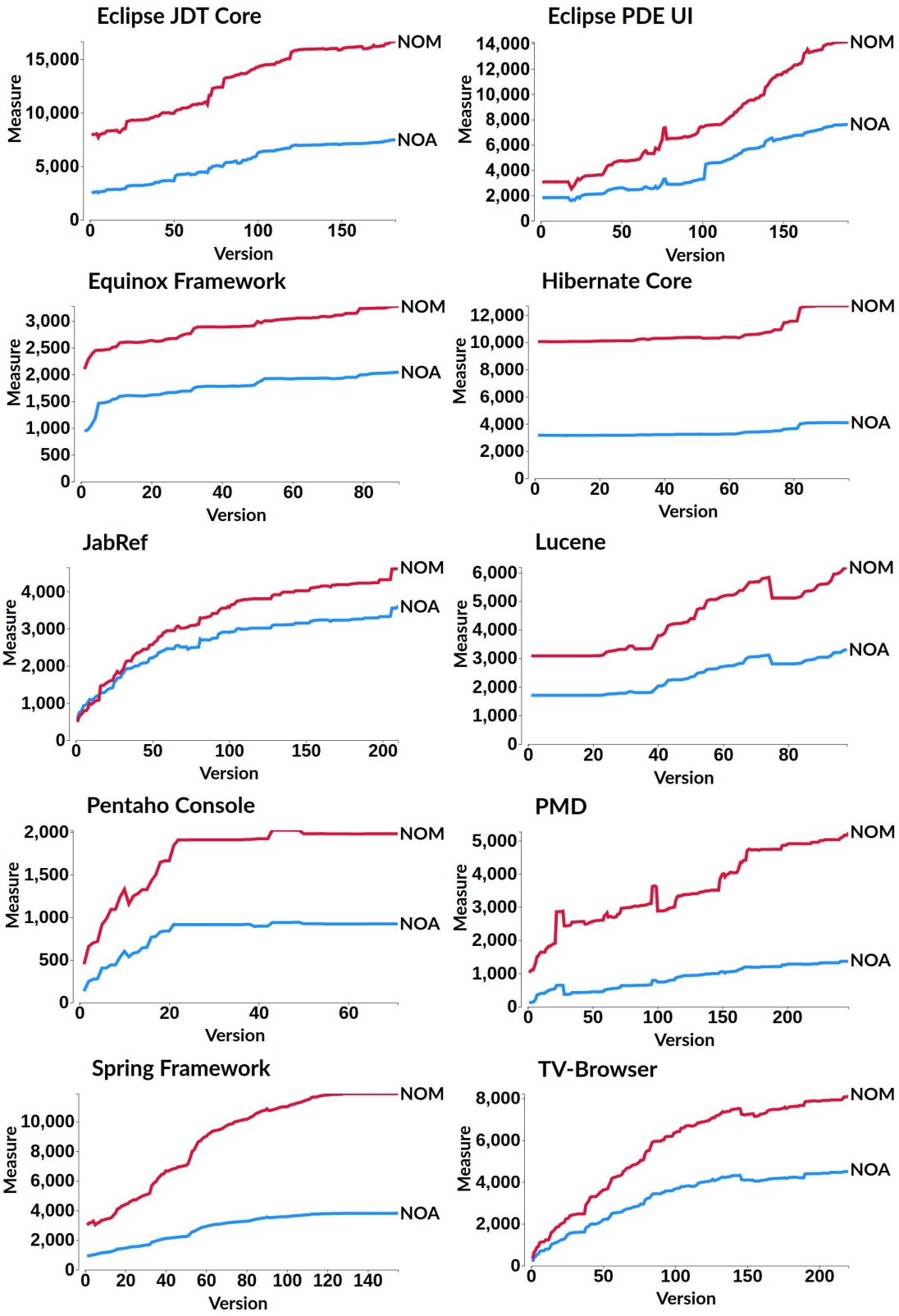


Figure 5.6: Global NOA/NOM time series of the analyzed systems.

Table 5.13: $\overline{R}^2$ values computed from the NOM models.

| System | lin. | quad. | cub. | log. 1 | log. 2 | log. 3 |
|---|---|---|---|---|---|---|
| Eclipse JDT Core | 99.80% | 99.81% | 99.81% | 99.76% | 99.77% | 99.77% |
| Eclipse PDE UI | 99.85% | 99.85% | 99.85% | 99.78% | 99.80% | - |
| Equinox Framework | 99.29% | 99.31% | 99.31% | 99.30% | 99.33% | 99.33% |
| Hibernate Core | 98.61% | 98.67% | - | 98.71% | 98.73% | - |
| JabRef | 99.84% | 99.84% | 99.86% | 99.67% | 99.67% | 99.70% |
| Lucene | 99.09% | 98.79% | 99.08% | 99.28% | 98.90% | 99.29% |
| Pentaho Console | 98.41% | 98.42% | 98.52% | 97.99% | 97.98% | 98.09% |
| PMD | 99.23% | 99.20% | 99.16% | 98.96% | 98.96% | 98.96% |
| Spring Framework | 99.93% | 99.94% | 99.91% | 99.88% | 99.88% | 99.89% |
| TV-Browser | 99.92% | 99.93% | 99.93% | 99.50% | 99.69% | 99.80% |

Observing the obtained results for NOA in Table 5.12, we note that almost all models presented $\overline{R}^2$ scores higher than or equal to 90%. We identify only two exceptions to which there is no relevant model, the cubic and logarithmic at degree 3 models for the global time series of `Hibernate Core`. Due to this, we did not highlight these cases with green in Table 5.12. For the initially selected models, we observe that linear, quadratic, logarithmic at degree 1, and logarithmic at degree 2 described well all the systems' global time series and attended to the Stage 2 of our protocol. However, applying the simplicity criteria, we conclude that the linear model is the one that better explains the growth evolution pattern of NOA since it has attended all aspects of our evaluation protocol.

In the case of NOM, reported in Table 5.13, we also identify that most of the produced models had good $\overline{R}^2$ values. Three cases were selected at Stage 1 because our method could not find models representing them: cubic and logarithmic at degree 3 for `Hibernate Core` and logarithmic at degree 3 for `Eclipse PDE UI`. Due to this, we did not highlight these cases with green in Table 5.13. Among the selected cases at Stage 1, we realize that linear, quadratic, logarithmic at degree 1, and logarithmic at degree 2 models represented well all systems' global time series. However, following Stage 3 of our evaluation protocol, we conclude that as well as for NOA, the linear model is the one that better explains the global evolution pattern of NOM.

**Summary of RQ1 - Size:** The number of attributes and methods of classes increases over the software evolution by following a linear model. Tables 5.12 and 5.13 show that the linear model returned $\overline{R}^2$ scores higher than 95% for all analyzed time series in both metrics. Such values indicate that this type of model is very efficient in describing the growth behavior of NOA and NOM over time. Besides, the number of methods is usually much higher than the number of attributes in terms of absolute values.

### 5.5.2 Evolution of NOA/NOM Relation

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

With RQ2, we aim to analyze 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 in two ways: global and class-by-class. The global proportion consists of summing the NOA and NOM values for the systems' classes and extracting a global value for each system's version. After that, we divided the global NOA by the global NOM, and then, we obtained the global proportion of these metrics. Figure 5.7 summarizes the global proportion of NOA and NOM extracted from the systems.

Moreover, the class-by-class proportion comprises computing the proportion for each class's version from the systems by dividing their NOA by NOM measures. After identifying the individual proportion, we computed the arithmetic average of each systems' version to find the average proportion of the classes in each systems' version. We show the class-by-class proportion of NOA and NOM of each system in Figure 5.8.

Analyzing Figure 5.7, we observe that the global NOA and NOM proportions at the beginning of the analyzed systems' life cycle are small, approximately 30%, and have increased over the systems' evolution. We identify this behavior in four systems: `Eclipse JDT Core`, `Equinox Framework`, `Pentaho Console`, and `PMD`. This finding shows that although the number of methods is remarkably higher than the number of attributes in terms of absolute values, the proportion of attributes concerning the number of methods tends to increase over time, i.e., the number of attributes grows in a higher rate than the number of methods. We also identify three cases where these proportions differ little over the evolution and remain visually stable. The systems with this behavior are `Hibernate Core`, `Lucene`, and `Spring Framework`. On the other hand, `JabRef`, `Eclipse PDE UI`, and `TV-Browser` had this proportion decreased over

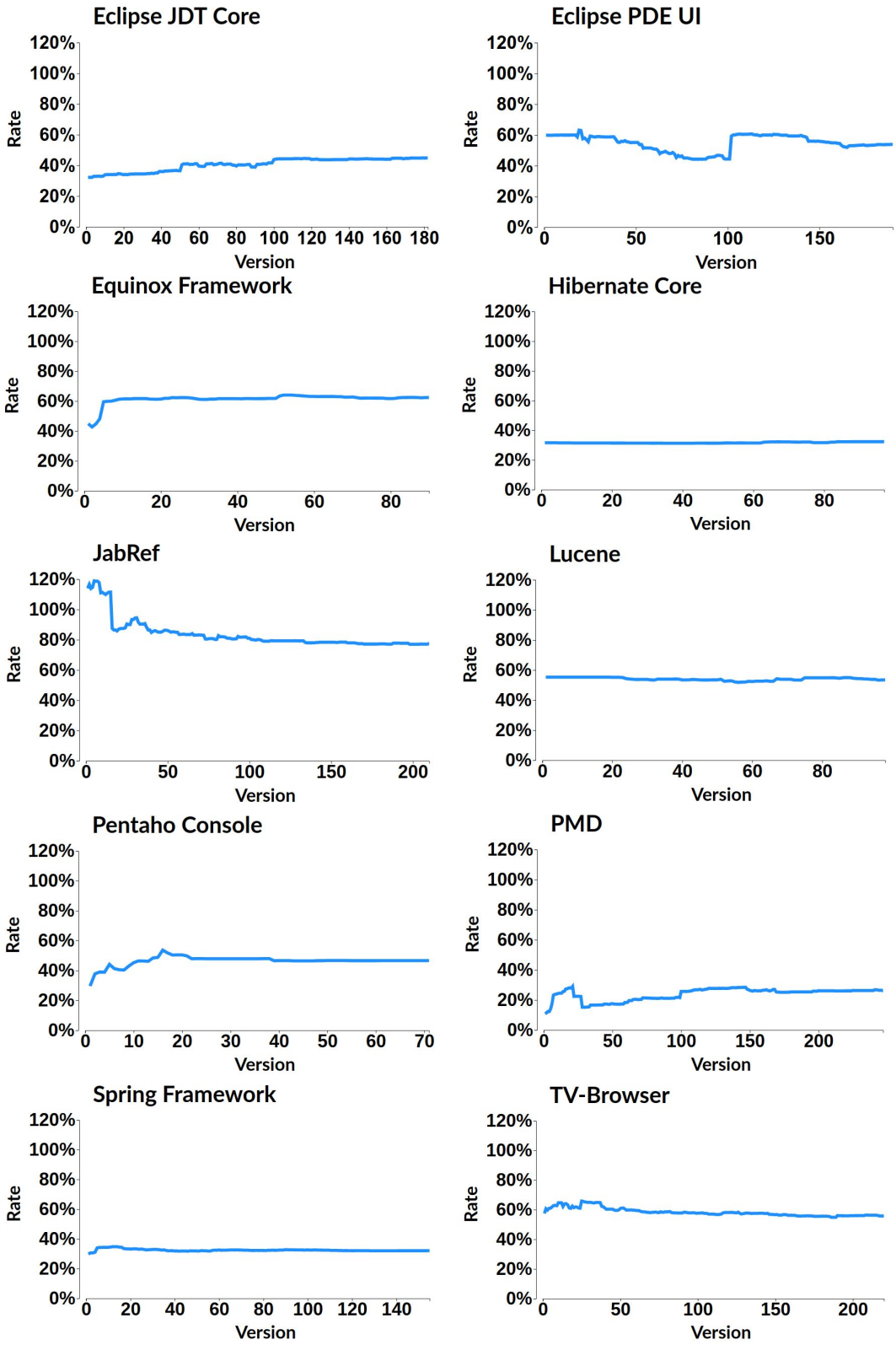


Figure 5.7: Global NOA/NOM proportion.

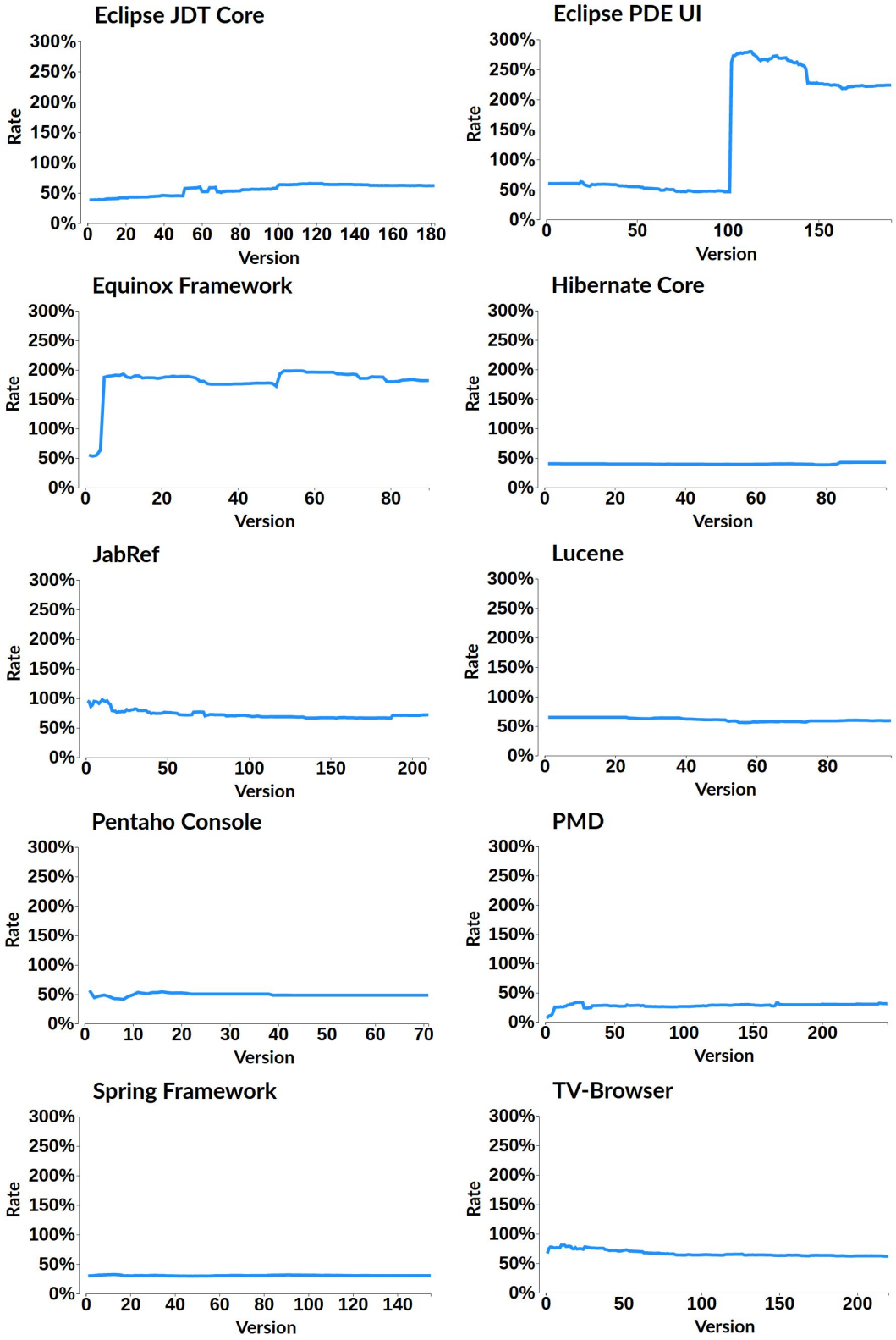


Figure 5.8: Arithmetic average of class-by-class NOA/NOM proportion.

time. Moreover, we identified that, in general, the global NOA and NOM proportion has varied in a very common interval over the software evolution, which goes from 30% to 60%.

The charts shown in Figure 5.8 indicate that the evolution of NOA/NOM in the analysis class-by-class follows the same pattern of the global analysis, shown in Figure 5.7. This finding reinforces the idea that the number of attributes grows at a higher rate than the number of methods. However, although the charts in Figures 5.7 and 5.8 follow the same pattern, we may observe that for some systems such as `Eclipse PDE UI` and `Equinox Framework`, the percentages obtained in Figure 5.8 are higher than the ones identified in Figure 5.7. As Figure 5.8 refers to the arithmetic average of class-by-class NOA and NOM percentages, we can conclude that, in these systems, most of the classes have more attributes than methods, and there is a low number of classes with more number of methods than the number of attributes.

**Summary of RQ2 - Size:** Although the number of methods is much higher than the number of attributes in terms of absolute values, in general, the number of attributes tends to increase in a higher rate than the number of methods over the software evolution. The global proportion of NOA in relation to NOM grows varies from 30% to 60%r over the software evolution.

### 5.5.3 Size Growth/Decrease Analysis

This section answers **RQ3.** *What set of classes within the software system affects the dimensions of growth/decrease and how these classes evolve?*.

With RQ3, we aim to identify the percentages of classes existing in the systems that directly affect the class size growth or decrease. We performed the trend analysis considering the time series from the systems' classes regarding the NOA and NOM metrics. Besides, we computed the percentages of types responsible for increasing and decreasing these metrics values and summarized them in Figure 5.9.

Table 5.14 shows the class percentages that interfere in the NOA growth and decrease. We discuss the results obtained by growth and decrease of the size metrics in the sequel.

#### 5.5.3.1 Classes Responsible for Size Growth

Figure 5.9 and Table 5.14 show that the median of the class percentage responsible for "NOA growth" and "NOM growth" is 12.50% and 16.50%, respectively. The maximum

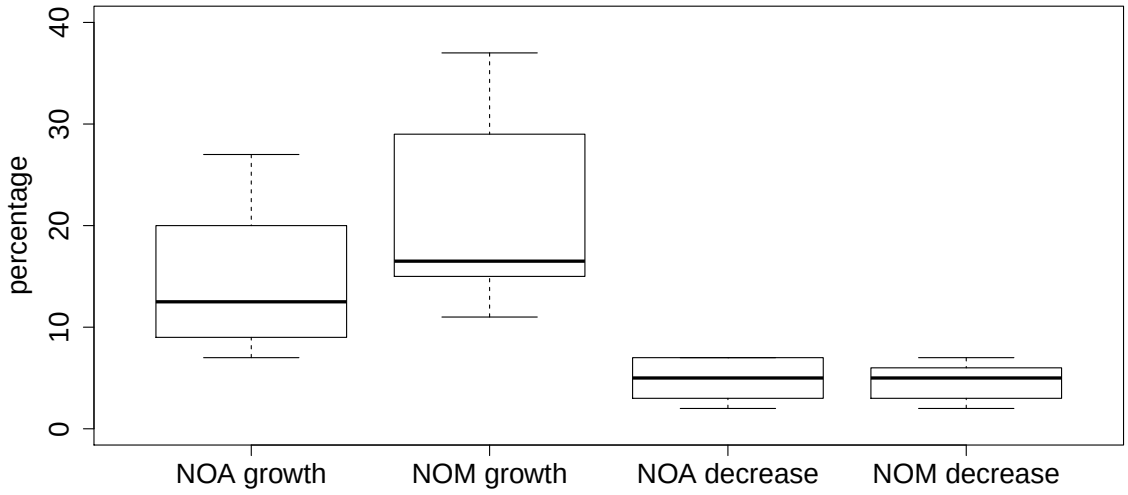


Figure 5.9: Distribution of classes that affects class size growth/decrease.

Table 5.14: Descriptive analysis of the distribution of classes that affects size growth/decrease.

| Event | 0% | 25% | 50% | 75% | 100% |
|---|---|---|---|---|---|
| NOA Growth | 7.00 | 9.00 | 12.50 | 19.25 | 27.00 |
| NOM Growth | 11.00 | 15.25 | 16.50 | 26.50 | 37.00 |
| NOA Decrease | 2.00 | 3.00 | 5.00 | 7.00 | 7.00 |
| NOM Decrease | 2.00 | 3.00 | 5.00 | 5.75 | 7.00 |

percentages indicate that 27.00% and 37.00% of the systems' classes contribute directly to increase NOA and NOM.

These results show that just a small group of classes in a system have their number of attributes and methods increased over time. The classes with increasing NOM correspond to no more than 37% of the system. Regarding NOA, this group is even smaller, with no more than 27% of the systems' classes.

#### 5.5.3.2 Classes Responsible for Size Decrease

Figure 5.9 and Table 5.14 show that the median of the class percentages responsible for "NOA decrease" and "NOM decrease" is 5.00%, and the maximum percentage is 7.00% for both. These results show that a tiny group of classes within the systems, corresponding to no more than 7.00%, directly contributes to these metrics decreasing over time. Although the class groups responsible for the growth and decrease of these metrics are small, the high discrepancy between them is one reason for the growth

trend identified for these metrics.

#### 5.5.3.3 Growth versus Decrease

In this analysis, we verified the intersection trend results for NOA and NOM to identify the percentage of classes that affects the following behaviors: (i) both NOA and NOM growth, (ii) both NOA and NOM decrease, (iii) NOA growth and NOM decrease, and (iv) NOA decrease and NOM growth. Initially, we analyzed these cases from the system perspective. i.e., considering percentages based on the total number of classes from the systems. Table 5.15 summarizes the results obtained for these cases.

Table 5.15: Intersection percentages of the trend results for NOA and NOM from the system perspective.

| System | i | ii | iii | iv |
|---|---|---|---|---|
| Eclipse JDT Core | 22% | 2% | 1% | 3% |
| Eclipse PDE UI | 7% | 3% | 1% | 2% |
| Equinox Framework | 11% | 1% | 1% | 0% |
| Hibernate Core | 7% | 1% | 0% | 0% |
| JabRef | 10% | 1% | 0% | 1% |
| Lucene | 7% | 1% | 0% | 1% |
| Pentaho Console | 5% | 1% | 0% | 1% |
| PMD | 7% | 1% | 1% | 0% |
| Spring Framework | 18% | 4% | 1% | 1% |
| TV-Browser | 16% | 3% | 1% | 1% |

As shown by the data in Table 5.15, the cases *(iii)* and *(iv)* are too rare, and their maximum percentages in the analysis are 1% and 3%, respectively. The Case *(i)* is the one with the highest chance of occurring since its maximum percentage corresponds to 22% of the total systems' classes. In Case *(ii)*, the maximum percentage is 4%.

Analyzing the intersection percentages from the perspective of the trend classes, we found the results from Table 5.16. To calculate these percentages, we divided the number of intersections by the number of classes that presented any trend for NOA or NOM.

Observing Table 5.16, we identify high percentages of Case *(i)*, which vary from ≈25% to ≈45%. These percentages had a significant increase in comparison to Table 5.15. For cases *(ii)* and *(iv)*, as in Table 5.15, Table 5.16 also indicates very low percentages. Such values show that these cases tend not to occur frequently during the evolution of the software. Therefore, the results suggest evidence of the relation between NOA and NOM over the software evolution. This evidence indicates that these software metrics grow together and, consequently, follow a combined pattern

Table 5.16: Intersection percentages of the trend results for NOA and NOM from the perspective of trend classes.

| System | i | ii | iii | iv |
|---|---|---|---|---|
| Eclipse JDT Core | 43% | 5% | 2% | 5% |
| Eclipse PDE UI | 28% | 11% | 2% | 7% |
| Equinox Framework | 38% | 4% | 3% | 2% |
| Hibernate Core | 35% | 6% | 2% | 1% |
| JabRef | 39% | 4% | 2% | 6% |
| Lucene | 31% | 3% | 1% | 3% |
| Pentaho Console | 24% | 6% | 1% | 4% |
| PMD | 33% | 5% | 3% | 1% |
| Spring Framework | 42% | 9% | 1% | 3% |
| TV-Browser | 40% | 9% | 2% | 3% |

over evolution.

**Summary of RQ3 - Size:** A small but not irrelevant percentage of classes in a system has the attributes and methods increased over time. Decreasing methods or attributes are rare events. Analyzing the systems as a whole, on average, just ≈10% of the classes in a system have methods and attributes growing together. However, considering only trend classes, NOA and NOM grow together.

## 5.6 Threats to Validity

This section presents the threats to the validity of this empirical analysis and discusses the main decision we made to mitigate them.

**Statistical trend tests.** To identify the percentage of classes that directly affect the growth and decrease of the software metrics, we defined a trend analysis with some statistical trend tests. If the choice of the trend tests is not well-planned, they may be considered a threat to validity since they may be susceptible to errors and return false-positives and false-negatives. We chose three relevant and useful trend tests to integrate our analysis and analyzed each time series by applying them to mitigate this threat. Besides, as a trend criteria, we defined that a trend exists in a time series when it is identified by at least two out of the three trend tests.

**Results generalization.** We analyzed the evolution of coupling, size, and inheritance hierarchy in open-source Java systems. We used a dataset composed of 10 different systems in our analysis. Although our dataset has an appropriated amount of

data, and reflect well the evolution of Java systems, we can not claim generalization of the results to other domains and contexts of development, such as proprietary software and systems written in any language other than Java.

**Software metrics chosen.** We chose software metrics proposed in the literature to investigate the evolution of coupling, inheritance hierarchy, and class size. The choice of these metrics may be considered a threat to validity if the selection is not well-planned, and the ones chosen do not provide a good representation of the analyzed characteristics. To mitigate this threat to validity, we used well-known metrics that have been applied in other studies in the literature.

**Ghost classes.** We performed a step to detect the time series of "ghost" classes during the trend analysis to avoid applying the trend tests in them. We did this because "ghost" classes have time series with broken intervals and do not present values for some observations. However, the removal of "ghost" classes may be considered a threat to validity since they contain relevant information about the trends. To mitigate this threat, we evaluated them separately to check if they do have relevant information. After assessing them, we concluded that they made up a tiny part of the systems and did not have significant trend patterns. Therefore, disregarding the application of the trend tests in the time series of "ghost" classes would not introduce bias in the analysis.

**Use of regression techniques.** To model the global time series of the systems, we defined a behavior analysis using linear regression techniques to define the models that describe the time series pattern. Although regression techniques have been very used to model time series [Graves et al., 2000; Ramil and Lehman, 2000; Capiluppi, 2003; Koch, 2005; Arisholm and Briand, 2006; Ratzinger et al., 2007; Shatnawi and Li, 2008; Kirbas et al., 2014], the presence of interventions or autocorrelation in the time series may bias the generated models. To avoid this problem, after generating the models using regression techniques, we carried out intervention and residual analyses to treat autocorrelation and, hence, ensure that the models have a good adjust to the global time series of the analyzed systems.

## 5.7 Final Remarks

This chapter presented and discussed the main observations extracted from some empirical analyzes about the evolution of object-oriented software systems from the perspective of coupling, size, and inheritance hierarchy. Our analysis considered a dataset composed of software metrics time series regarding 10 Java open-source systems. To study the evolution of these dimensions, we used fan-in/fan-out, DIT/NOC, and NOA/NOM

for representing coupling, inheritance hierarchy, and size, respectively. After mapping the dimensions into software metrics, we applied our two-phase analysis method, defined in Chapter 4, to analyze how these aspects evolve.

Based on these results, we depicted software evolution properties that we present in Chapter 6.

# Chapter 6

# Software Evolution Properties

This chapter compiles and discusses the results of the empirical analysis carried out in this study. The results lead us to identify 15 software evolution properties related to coupling, inheritance, and size. We organize this chapter as follows. Section 6.1 shows the properties of coupling evolution. Section 6.2 presents the properties of inheritance hierarchy evolution. Section 6.3 describes the ones of size evolution, and Section 6.4 concludes this chapter.

## 6.1 Coupling

This section presents seven evolution properties regarding coupling evolution. They are:

**1st - Coupling grows linearly over time.** We have identified that the global fan-in and fan-out time series are better modeled by a linear model. The increasing level of coupling among classes may make the system harder to comprehend and maintain. Therefore, this property is under Lehman's $2^{nd}$ and $6^{th}$ laws, which indicate that the complexity tends to increase, and the quality tends to decline during the software evolution. We consider the more likely cause of this property is the inclusion of new features in the system without proper adjustments to keep coupling stable. Nevertheless, this assumption needs investigation.

**2rd - 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. Unnecessary coupling is much higher than necessary coupling since the first releases 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, even more care should be given for classes that provide services as the impact of changes on them tends to be higher over time.

**3th - A small group of classes have high coupling.** There is a small group of classes in a system that have their coupling level increased or decreased. In our analysis, we found that no more than 35% of classes in a system have their coupling level increased, and no more than 10% of classes have their coupling decreased.

**4th - Complexity is introduced since the first versions of a system.** We observe that since the first system versions, the unnecessary coupling is hugely higher than the necessary coupling. Based on this analysis and the quality indication pointed by the literature, we conclude that the system's initial version is already complex. This finding contradicts the assumption that complexity is inserted in software systems over its evolution. However, this finding is accordance with the study of Tufano et al. [2015] whose main conclusion is that bad-smells are introduced in software systems since their first versions. This property indicates the need of developing refactoring tools and techniques to be applied since the beginning of the systems' life cycle.

**5th - Legacy classes mainly contribute to coupling evolution.** Legacy classes are classes introduced in the first system version, which are not removed during its evolution. Analyzing the the classes that influence the coupling growth/decrease, we identified a strong presence of legacy classes. This property corroborates that complexity is introduced in software systems since the beginning.

**6th - There is no association between fan-in and fan-out.** By analyzing the percentages of four association cases, we do not identify a pattern in terms of growth and decrease of fan-in and fan-out for the classes that impact the coupling evolution. Such finding suggests that the growth/decrease of fan-in is not related to the growth/decrease of fan-out. When a class becomes more or less dependent on other classes, it does not necessarily imply that the other classes will demand more or fewer services. However, to prove this property and the evidence obtained in it, we will evaluate the relationship between these metrics by applying a statistical test. With this test, we will be able to claim whether they are associated based on statistical significance.

## 6.2 Inheritance Hierarchy

Results of this work suggest five properties of inheritance hierarchy evolution:

**1st - Inheritance hierarchy tends to increase in depth and decrease in breadth over time.** We represent depth and breadth by the DIT and NOC metrics,

respectively. Their global time series analysis indicates that the mean DIT of classes within a system tends to increase, whereas the mean NOC tends to decrease.

**2nd - Inheritance hierarchy depth grows according to a linear model.** An inheritance tree with many levels may introduce complexity in the system structure and make it hard to understand and maintain. This fact is because the higher number of superclasses a class has, the more difficult it is to understand the behavior of its objects. Due to this, Gamma et al. [1994] define the principle *"favor composition over inheritance"* that recommends implementing reusable software using class composition rather than class inheritance to avoid such complexity.

**3rd - Inheritance hierarchy breadth decreases according to a quadratic model.** A quadratic function better models the global behavior of the breadth evolution in the inheritance hierarchy. We used the NOC metric to evaluate breadth in the inheritance hierarchy, and their global time series have several fluctuations between the versions. Therefore, we believe the quadratic model adjusted better to these time series since it has a curve more flexible than the other models. Decreasing the mean number of children in a system may be due to two possible reasons: the new classes added to the system do not have children classes in general and/or the inheritance trees are refactored over time.

**4th - A small part of the system influences the growth and the decrease of the inheritance hierarchy.** A small group of classes present in a system affects the growth and decrease of depth and breadth in the inheritance hierarchy. Regarding depth, no more than 21.00% of the system's classes have their DIT increased, and no more than 17.00% have their DIT decreased. Regarding breadth, this percentage is even less. No more than 8.00% and 2.00% of the systems' classes have their DIT increased or decreased. Therefore, although the use of inheritance may introduce complexity in the system, it will occur with a small portion of the system.

**5th - There is no association between the depth and the breadth of a class.** A small percentage of classes within the systems presented an association between depth and breadth in terms of growth/decrease. This finding shows that the number of children and the number of superclasses of a class evolve independently, and do not tend to relate or follow the combined pattern over time. We will re-evaluate this evidence latter by applying a statistical test and, consequently, claim with a significance level the presence of association or non-association between them.

## 6.3 Size

We identified following properties of class size evolution:

**1st - The size of classes grows according to the linear model.** A linear function better models the global pattern evolution of both the number of attributes and number of methods of classes within the systems. This property is in accordance of the Lehman's $6^{th}$ law, which indicates that the systems continually increase over time. Our results indicate how such increases occur in the structure of classes.

**2nd - The proportion of the number of attributes in relation to the number of methods in a class grows over time** We identified that both global and class-by-class NOA/NOM ratio grows over time. The NOA/NOM proportion tends to vary from 30% to 60%. This finding shows that although the number of methods is higher than the number of attributes in terms of absolute values, the growth of the number of attributes is higher than the number of methods in a system.

**3th - A small group of classes affects the growth of system size.** In our study, there is a small group of classes in the systems with an increasing number of attributes, no more than 27.00%. In the same way, few types have an increasing number of methods, up to 37.00%. Rarely, a class has its size decreased. In our study, 7.00% is the higher percentage of classes with such behavior we found. Although the percentage of types having attributes or methods added to them is not very high, it is still relevant. Apart from refactoring, a class swelling means that more services were introduced in the class. This fact may lead to non-focused and more complex classes and, therefore, to a more complex software structure.

**4th - The evolution of the number of attributes and the number of methods are correlated.** Our results suggest a positive correlation between NOA and NOM evolution regarding growth and decrease. Such finding means that when we include attributes in a class, methods are also included, or vice-versa. When removing attributes, methods are also excluded. This finding details the way a class grows or decreases over the software evolution. However, we evaluated the relation between NOA and NOM by descriptive analysis and identified evidence of an association between them. To confirm and prove this property, we will analyze it using a statistical test and, then we will validate this relation based on a statistical significance.

## 6.4 Final Remarks

This chapter summarized and discussed the results of the empirical analyzes we carried out in Chapter 5. Based in the results, we defined evolution properties for coupling,

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 dimensions.

Chapter 7 presents the next steps that we will follow to conclude this thesis research and provides some directions about the methodology we will perform to achieve the objectives of this work.

# Chapter 7

# Next Steps

This chapter presents the next steps to conclude this Ph.D. research, as well as the methodology we will apply in the work. We organize this chapter as follows. Section 7.1 describes a proposal of methodology to create a dataset with data of software evolution. Section 7.2 details a prototype of methodology for defining our prediction method for metrics values over the software evolution. Section 7.3 enlists the tasks we planned to achieve the proposed objectives and a schedule with the deadlines to meet each of the defined tasks. Section 7.4 concludes this chapter.

## 7.1 Creation of a Dataset

This section presents a proposal of the methodology we defined to create our software evolution dataset. It consists of six steps, which we describe as follows.

**Step 1 - Definition of software systems.** The first is to define the object-oriented software systems from which we will collect the data about evolution. We chose including only systems developed in Java in our dataset because this language is one of the most popular object-oriented programming languages in the academy and industry. Besides, there are many software systems developed in Java and many software metrics tools support the collection of metrics from systems created in this language.

Initially, we aim to start creating our dataset by extracting recent data from the systems existing in `COMETS`. We will decide throughout the conduction of the dataset extension if it is worth it to include data from other software systems and which new systems may be incorporated. The resulting data set will be used in the sequence of this work.

**Step 2 - Extract the software releases from their repository.** We will identify the repository of the systems in version control platforms, such as SVN and Git, and extract versions of their source code within a period. During this versioning process of the source code, we will stipulate a time frame of the systems life cycle that we will extract. After that, we will space out it in intervals of bi-weeks, as well as Couto et al. [2013] did in `COMETS`, to compose the many systems releases. Therefore, each version of the systems will refer to a period of 14 days within the defined total time frame.

**Step 3 - Identify tools for collecting software metrics.** This step aims to define the software metrics that we will collect from the systems of our dataset and search tools that support the collection of these metrics. After including the software metrics existing in `COMETS`, we will search by tools that may provide measures for them. However, during the dataset creation, we will decide if we will collect other object-oriented metrics.

**Step 4 - Collect the measures from the versions of the software systems.** In this step, we will run the chosen software tools to extract their from the systems' versions obtained in Step 2.

**Step 5 - Organize the collected metrics as time series.** In this step, we will to organize the metrics in *CSV* files for each metric and storing all metrics files regarding a given system into a *ZIP* file. To create the *CSV* files, we will follow the same pattern defined by Couto et al. [2013]. The metrics file lines will represent the systems' classes, whereas the columns will specify the releases we extracted from the programs. Each cell *(c, x)* into a *CSV* file will indicate the value of a particular metric $M$ computed for the class $c$ in a version $x$.

## 7.2 Prediction Method

This section describes the methodology to define our prediction method for software evolution. Chapter4 described an approach to analyze the behavior of a software time series (Section 4.1) as Phase 1 of our method of software evolution data analysis. This approach used linear regression and other techniques, such as intervention analysis and residuals autoregression, to improve the produced models' adjustment and treat external events that could distort the time series pattern and, consequently, influence the time series representation. It allowed us to represent and model a particular time series, but the evaluation protocol did not assess the produced models from the prediction point of view.

Therefore, the purpose here is to extend the approach described in Phase 1 of our analysis method by including a prediction evaluation protocol. This protocol will allow the method to only analyze the time series behavior up to the last available version, to produce a forecast model for the specific time series. Our method's input will take a set of evolutionary data regarding software metrics from a given software system, i.e., the metric time series. The result of the method will be a model that predicts how a software system will evolve in terms of the attributes measured by the metrics. To achieve this goal, we will follow four steps described as follows.

**Step 1 - Use of the behavior analysis approach as the foundation.** This step consists of reusing the steps 1 to 5 of the behavior analysis approach, proposed in Section 4.1 (Chapter4), as the base to build the initial prediction models for the analyzed time series. In the end, we will have a set of best models, each referring to the following types: linear, quadratic, cubic, logarithmic at Degree 1, logarithmic at Degree 2, and logarithmic at Degree 3.

**Step 2 - Definition of a selection strategy for prediction models.** It consists of defining a protocol for evaluating the forecast provided by the prediction models produced in Step 1 and selecting the best prognosis. To assess the models in this step, we will divide the time series presented by the developer into two groups: "training" and "test". The "training" set is the one we will use in Step 1 to generate the models, while we will consider the "test" set to analyze the models' prediction quality.

**Step 3 - Automation of the selection strategy.** This step aims to automate the production and choice of the best prediction model for the analyzed software time series. At the current stage of this research, we have this method partially automated in R programming language since we implemented the behavior analysis approach to carry out the empirical studies presented in Chapter 5. To make our prediction method completely automatic, we will incorporate the selection strategy for prediction models, which we will define in Step 2, to the R scripts we have already created.

**Step 4 - Evaluation of the prediction method.** We will carry out empirical studies aiming to evaluate the predicting model. For this purpose, we will consider in this step the metrics time series of the software systems regarding the dataset that we will also build as continuity of the current research described in this thesis project.

## 7.3 Schedule

This section presents a task schedule that we need to do to conclude this Ph.D. thesis. We divided the task into the remaining period of the Ph.D. course, which goes from

September $1^{st}$, 2020 to August $1^{st}$, 2021. We enlist the main tasks that still need to be done as follows.

1. Define the software systems we will include in the dataset.
2. Identify the defined systems in the version control platforms and extract their versioned source code.
3. Seek for software metrics tools in the literature that collects measure Java object-oriented systems by their source code.
4. Collect the metrics values from the systems releases extracted from the version control platforms.
5. Organize the collected metrics as time series in *CSV* files.
6. Construct a website for the dataset and make it available online for other researchers.
7. Write a paper about the proposed dataset.
8. Refine the methodology for creating our prediction method.
9. Define a selection strategy that allows our method to choose the best prediction models for each analyzed time series.
10. Implement our prediction method and make it completely automatic.
11. Evaluate the effectiveness of the models produced by our method.
12. Write a paper about the prediction model.
13. Write the Ph.D. thesis.
14. Present the final Ph.D. thesis.

Besides defining the task that still needs to be fulfilled, we stipulated deadlines for finishing each of them. Table 7.1 shows these deadlines.

Table 7.1: Task schedule for finishing the Ph.D. thesis.

| Task | Period | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 2020 | | | | 2021 | | | | | | | |
| | 09 | 10 | 11 | 12 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 |
| **1** | X | | | | | | | | | | | |
| **2** | X | X | | | | | | | | | | |
| **3** | | X | | | | | | | | | | |
| **4** | | | X | X | | | | | | | | |
| **5** | | | | X | | | | | | | | |
| **6** | | | | X | X | | | | | | | |
| **7** | | | | | | X | | | | | | |
| **8** | | | | | | | X | | | | | |
| **9** | | | | | | | X | | | | | |
| **10** | | | | | | | X | X | | | | |
| **11** | | | | | | | | X | X | | | |
| **12** | | | | | | | | | X | | | |
| **13** | X | X | X | X | X | X | X | X | X | X | X | |
| **14** | | | | | | | | | | | | X |

## 7.4 Final Remarks

This chapter presented the next steps that we proposed to conclude this Ph.D. thesis and detailed a proposal of methodology for each one of the goals that we aim to achieve. Besides, it also divided the next steps in a sequence of minor tasks and scheduled the deadlines for meeting each of them, considering the remaining period for finishing the Ph.D. course.

Chapter 8 concludes the thesis project by summarizing all content described in this document, highlighting the main contributions of this research, and presenting an overview of the next steps to conclude this research.

# Chapter 8

# Conclusion

The evolution process is one of the most critical phases in the software life cycle. It is responsible for most of the total cost of a system that may comprise from 85% to 90% of the total expenses that a company spends with software. 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 to reduce software costs. Lehman et al. [1997] carried out 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, much work have been done aiming to extract more details about the evolution process in software and validate the presence of these laws in the development context. The present work is concerned with object-oriented software evolution.

Initially, we carried out a Systematic Literature Review (SLR) to compile the current state-of-the-art on software evolution. We also aimed to identify researches opportunities that need to be performed to cover the still open points. Our analysis identified a total of 130 papers published from 1979 to 2019. The SLR revealed that software evolution has been studied from five perspectives:

**(i)** verification of the applicability of Lehman's laws;

**(ii)** proposal for applications;

**(iii)** analysis of the evolution with a focus on quality

**(iv)** analysis of the software structure evolution;

**(v)** proposal of models for software evolution.

Our SLR has also identified unusual characteristics of software evolution, such as the confirmation of increasing complexity and challenging maintenance over the software life cycle. However, we realized that the studies carried out have neither provided patterns explaining how the structure of software systems degrades over time nor have explained how it evolves from the perspective of the software's internal dimensions.

This work aims to contribute with a solution to this problem in two directions. First, we analyzed the evolution of the object-oriented software systems aiming to provide a fine-grained view of how software internal structure evolves from the perspective of coupling, inheritance hierarchy, and size of classes. We defined a novel method based on time series analysis, linear regression techniques, and trend tests to analyze the evolution of object-oriented systems. This method consists of two phases. The first phase models the evolutionary data extracted from the software using linear regression to represent their behavior and evolution pattern. The second phase applies trend tests in time series regarding the software measures for obtaining the internal components that directly impact the increase or decrease of software measures.

Applying such an approach in evolutionary data from ten Java-based open-source projects, we extracted a set of properties that describe and characterize the software systems' evolution behavior from the perspective of the internal dimensions. Some of the properties we defined are: (i) the coupling, size of classes and the depth of the inheritance hierarchy increase according to the linear model; (ii) the breadth of the inheritance hierarchy decreases according to a quadratic model; (iii) systems are designed with a high level of complexity; (iv) most of the systems' classes do not change their metrics values over time and, consequently, the increase or decrease of dimension in the system is directly affected by a small group of classes.

In the second direction, we aim to define a prediction method for object-oriented software evolution applying the software evolution properties identified in this thesis project as a background in the. Our approach will build forecast models for particular systems considering data of coupling, size of classes, and inheritance hierarchy. To create this method, we will base on our analysis approach defined to analyze the evolution of the software dimension and extend it to make the produced models predict future values of the examined aspects. Our method will take a set of evolutionary data regarding software metrics from a particular system as input. It will return a model that project values for future releases of that software based on the behavior extracted for its metrics history. By this approach, developers will be able to produce specific prediction models for a particular software. These models will aid them to better plan their strategies for making changes and new features in the system and, then, avoid or mitigate the software architecture degrading over their evolution.

To evaluate the proposal, we will extend the software evolution dataset, `COMETS`, which we used in the empirical analysis in the first part of this research. It contains a large quantity of evolutionary data of software metrics extracted from ten object-oriented software. However, it is out of date. The most recent information existing in it refers to December $11^{st}$, 2011. Then, we aim to mine the values of software metrics from recent releases of their systems and use these data to assess our prediction method. It is essential to highlight that the prediction method and the extended dataset will be built in the next steps of this thesis research.

The results obtained so far in this thesis project have produced the following contributions:

- A Systematic Literature Review that compiles the knowledge on software evolution. The results of the SLR is important for industry and academy. In the industry side, such knowledge will aid decision making in the software life cycle. In the academy side, the results help to identify the need for further research based on the knowledge the community has so far on software evolution.

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

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

Besides, the research carried out until now in this thesis project has generated the following scientific papers:

- 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)

- 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 Software Components, Architectures, and Reuse (SBCARS '19), Salvador, BA,

Brazil, pages 23–32, 2019. (Published and awarded as the 2*nd* best paper in the symposium)

- Sousa, B.L.; Bigonha, M.A.S; Ferreira, K.A.M.; Franco G.C. Evolution of Size and Inheritance in Object-Oriented Software – A Time Series Based Approach. Submitted to an international journal, pages 1–11, 2020. (Under Review)
- Sousa, B.L.; Bigonha, M.A.S; Ferreira, K.A.M.; Franco G.C. A Comprehensive Systematic Literature Review of Software Evolution. Submitted to an international journal, pages 1–35, 2020. (Under Review)

As future works and the next steps for this Ph.D. research, we intend:

- Built a dataset with recent data regarding the evolution of object-oriented software systems.
- Define a method that extracts prediction models from the evolutionary data regarding software metrics for object-oriented software systems.
- Evaluate the proposed prediction method considering real data of software evolution extracted from object-oriented software systems.

# Bibliography


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

Al-Ajlan, A. (2009). The evolution of open source software using eclipse metrics. pages 211–218.

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

Alenezi, M. and Zarour, M. (2015). Modularity measurement and evolution in object-oriented open-source projects. volume 24-26-September-2015.

Alnaeli, S. M., Taha, A. D. A., and Timm, T. (2016). On the prevalence of function side effects in general purpose open source software systems. pages 141 – 148, Baltimore, MD, United states.

Antinyan, V., Staron, M., Meding, W., Österström, P., Bergenwall, H., Wranker, J., Hansson, J., and Henriksson, A. (2013). Monitoring evolution of code complexity in agile/lean software development: A case study at two companies. pages 1–15.

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

Antoniol, G., Villano, U., Merlo, E., and Di Penta, M. (2002). Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44(13):755–765.

Arisholm, E. and Briand, L. (2006). 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.

Arisholm, E., Briand, L. C., and Foyen, A. (2004). Dynamic coupling measurement for object-oriented software. *IEEE Transactions on software engineering*, 30(8):491--506.

Baer, N. and Zeidman, R. (2009). Measuring software evolution with changing lines of code. pages 264 – 270, New Orleans, LA, United states.

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

Barry, E. J., Kemerer, C. F., and Slaughter, S. A. (2003). On the uniformity of software evolution patterns. In *Proceedings of the 25th International Conference on Software Engineering*, pages 106--113. IEEE Computer Society.

Bavota, G., Canfora, G., Di Penta, M., Oliveto, R., and Panichella, S. (2013). The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *2013 IEEE international conference on software maintenance*, pages 280--289. IEEE.

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

Beyer, D. and Hassan, A. (2006). Animated visualization of software history using evolution storyboards. pages 199–208.

Bhattacharya, P., Iliofotou, M., Neamtiu, I., and Faloutsos, M. (2012). Graph-based analysis and prediction for software evolution. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 419–429. ISSN 1558-1225.

Biolchini, J., Mian, P. G., Natali, A. C. C., and Travassos, G. H. (2005). Systematic review in software engineering. *System Engineering and Computer Science Department COPPE/UFRJ, Technical Report ES*, 679(05):45.

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

Bowerman, B. and O'Connell, R. (1993). *Forecasting and Time Series: An Applied Approach*. Duxbury classic series. Duxbury Press. ISBN 9780534932510.

Box, G. and Jenkins, G. (1976). *Time Series Analysis: Forecasting and Control*. Holden-Day, San Francisco.

Bredeweg, B. and Struss, P. (2003). Current topics in qualitative reasoning. *AI Magazine*, 24(4):13--13.

Breiman, L. (2017). *Classification and regression trees*. Routledge.

Burd, E. and Munro, M. (1999). Initial approach towards measuring and characterizing software evolution. *Reverse Engineering - Working Conference Proceedings*, pages 168 – 174.

Businge, J., Serebrenik, A., and van den Brand, M. (2010). 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.

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

Capiluppi, A., Fernandez-Ramil, J., Higman, J., Sharp, H., and Smith, N. (2007). 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.

Capiluppi, A., Morisio, M., and Ramil, J. (2004a). 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.

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

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

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

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

Chatzigeorgiou, A. and Manakos, A. (2014). Investigating the evolution of code smells in object-oriented systems. *Innovations in Systems and Software Engineering*, 10(1):3–18.

Chen, T., Gu, Q., Wang, S., Chen, X., and Chen, D. (2008). Module-based large-scale software evolution based on complex networks. In *2008 8th IEEE International Conference on Computer and Information Technology*, pages 798--803. IEEE.

Chevalier, F., Auber, D., and Telea, A. (2007). Structural analysis and visualization of c++ code evolution using syntax trees. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 90--97. ACM.

Chhabra, J. K. and Gupta, V. (2010). A survey of dynamic software metrics. *Journal of computer science and technology*, 25(5):1016--1029.

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

Collberg, C., Kobourov, S., Nagra, J., Pitts, J., and Wampler, K. (2003). A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 77--ff. ACM.

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

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

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

D'Ambros, M. and Lanza, M. (2008). A flexible framework to support collaborative software evolution analysis. In *2008 12th European Conference on Software Maintenance and Reengineering*, pages 3--12. IEEE.

D'Ambros, M., Lanza, M., and Lungu, M. (2006). The evolution radar: Visualizing integrated logical coupling information. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 26--32. ACM.

D'Ambros, M., Lanza, M., and Robbes, R. (2010). An extensive comparison of bug prediction approaches. In *MSR 2010*, pages 31--41. IEEE.

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

Decan, A., Mens, T., and Grosjean, P. (2019). An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24(1):381–416.

Digkas, G., Lungu, M., Chatzigeorgiou, A., and Avgeriou, P. (2017). The evolution of technical debt in the apache ecosystem. volume 10475 LNCS, pages 51 – 66, Canterbury, United kingdom. ISSN 03029743.

Draper, N. and Smith, H. (1981). *Applied Regression Analysis*. Applied Regression Analysis. Wiley. ISBN 9780471029953.

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

Feitelson, D. (2012). Perpetual development: A model of the linux kernel life cycle. *Journal of Systems and Software*, 85(4):859–875.

Fernandez, A. and Bergel, A. (2018). A domain-specific language to visualize software evolution. *Information and Software Technology*, 98:118–130.

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

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

Ferreira, K. A. M., Moreira, R. C. N., Bigonha, M. A. S., and Bigonha, R. S. (2012). The evolving structures of software systems. In *WETSoM*, pages 28--34. ISSN 2327-0950.

Filó, T. (2014). Identification of thresholds for metrics of oriented-object software. Master's thesis, UFMG, Belo Horizonte, Minas Gerais,. (In Portuguese).

Filó, T., Bigonha, M., and Ferreira, K. (2015). A catalogue of thresholds for object-oriented software metrics. In SOFTENG, editor, *Proceedings of International Conference on Advances and Trends in Software Engineering*, page 1.

Fischer, M. and Gall, H. (2006). Evograph: A lightweight approach to evolutionary and structural analysis of large software systems. pages 179 – 188, Benevento, Italy. ISSN 10951350.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Gall, H., Jazayeri, M., Klosch, R. R., and Trausmuth, G. (1997). Software evolution observations based on product release history. In *1997 Proceedings International Conference on Software Maintenance*, pages 160--166. IEEE.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition. ISBN 0201633612.

German, D., Adams, B., and Hassan, A. (2013). The evolution of the r software ecosystem. pages 243–252.

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

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

Gonzalez-Barahona, J., Robles, G., Herraiz, I., and Ortega, F. (2014). Studying the laws of software evolution in a long-lived floss project. *Journal of Software: Evolution and Process*, 26(7):589--612.

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

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

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

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

Häggström, O. et al. (2002). *Finite Markov chains and algorithmic applications*, volume 52. Cambridge University Press.

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

Hassoun, Y., Counsell, S., and Johnson, R. (2005). Dynamic coupling metric: proof of concept. *IEE Proceedings-Software*, 152(6):273--279.

Hassoun, Y., Johnson, R., and Counsell, S. (2004a). 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.

Hassoun, Y., Johnson, R., and Counsell, S. (2004b). Empirical validation of a dynamic coupling metric. *Birkbeck College London, Technical Report BBKCS-04-03*.

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

Hattori, L., D'Ambros, M., Lanza, M., and Lungu, M. (2013). Answering software evolution questions: An empirical evaluation. *Information and software technology*, 55(4):755--775.

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

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

Herraiz, I., Robles, G., Gonzalez-Barahona, J. M., Capiluppi, A., and Ramil, J. F. (2006). Comparison between slocs and number of files as size metrics for software evolution analysis. In *CSMR'06*, pages 206--213. ISSN 1534-5351.

Hindle, A., Jiang, Z. M., Koleilat, W., Godfrey, M. W., and Holt, R. C. (2007). Yarn: Animating software evolution. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 129--136. IEEE.

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

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

Kachigan, S. K. (1991). *Multivariate statistical analysis: A conceptual introduction*. Radius Press.

Kanda, T., Ishio, T., and Inoue, K. (2013). Extraction of product evolution tree from source code of product variants. pages 141 – 150, Tokyo, Japan.

Kanwal, J., Basit, H. A., and Maqbool, O. (2018). Structural clones: An evolution perspective. In *2018 IEEE 12th International Workshop on Software Clones (IWSC)*, pages 9--15. IEEE.

Kanwal, J., Maqbool, O., Basit, H., and Sindhu, M. (2019). Evolutionary perspective of structural clones in software. *IEEE Access*, 7:58720–58739.

Kemerer, C. F. and Slaughter, S. (1999). An empirical approach to studying software evolution. *IEEE transactions on software engineering*, 25(4):493--509.

Kendall, M. G. (1975). *Rank correlation methods*. Charless Griffin, LDN.

Khoshgoftaar, T. M., Allen, E. B., Jones, W. D., and Hudepohl, J. (1999). Classification tree models of software quality over multiple releases. In *Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No. PR00443)*, pages 116--125. IEEE.

Khoshgoftaar, T. M., Munson, J. C., and Lanning, D. L. (1993). Dynamic system complexity. In *[1993] Proceedings First International Software Metrics Symposium*, pages 129--140. IEEE.

Kikas, R., Gousios, G., Dumas, M., and Pfahl, D. (2017). Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 102--112. IEEE press.

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

Kitchenham, B. and Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering. Technical report EBSE 2007-001, Keele University and Durham University Joint Report.

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

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

Kpodjedo, S., Ricca, F., Galinier, P., Antoniol, G., and Guéhéneuc, Y.-G. (2013). Studying software evolution of large object-oriented software systems using an etgm algorithm. *Journal of software: Evolution and Process*, 25(2):139–163.

Krishnan, S., Lutz, R. R., and Goševa-Popstojanova, K. (2011a). Empirical evaluation of reliability improvement in an evolving software product line. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 103--112. ACM.

Krishnan, S., Strasburg, C., Lutz, R., and GoÅ¡eva-Popstojanova, K. (2011b). Are change metrics good predictors for an evolving software product line?

Langelier, G., Sahraoui, H., and Poulin, P. (2008). Exploring the evolution of software quality with animated visualization. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 13--20. IEEE.

Lanza, M. and Ducasse, S. (2002). Understanding software evolution using a combination of software visualization and software metrics. In *In Proceedings of LMO 2002 (Langages et Modèles à Objets*. Citeseer.

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

Lee, Y., Yang, J., and Chang, K. H. (2007b). Metrics and evolution in open source software. In *Seventh International Conference on Quality Software (QSIC 2007)*, pages 191--197. IEEE. ISSN 1550-6002.

Lehman, M. M. (1996). Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108--124. Springer.

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., and Turski, W. M. (1997). Metrics and laws of software evolution-the nineties view. In *Proceedings Fourth International Software Metrics Symposium*, pages 20--32. IEEE.

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

Li, H., Huang, B., and Lu, J. (2008). Dynamical evolution analysis of the object-oriented software systems. pages 3030–3035.

Li, H., Zhao, H., Cai, W., Xu, J.-Q., and Ai, J. (2013). A modular attachment mechanism for software network evolution. *Physica A: Statistical Mechanics and its Applications*, 392(9):2025--2037.

Li, W. (1999). Another metric suite for object-oriented programming. *J. Syst. Softw.*, 44(2):155–162. ISSN 0164-1212.

Lientz, B. P. and Swanson, E. B. (1980). *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., USA. ISBN 0201042053.

Liu, Y. and Ai, J. (2016). A software evolution complex network for object oriented software.

Longo, F., Tiella, R., Tonella, P., and Villafiorita, A. (2008). Measuring the impact of different categories of software evolution. In *Software Process and Product Measurement*, pages 344--351. Springer.

Lorenz, M. and Kidd, J. (1994). *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, Inc., USA. ISBN 013179292X.

Louridas, P., Spinellis, D., and Vlachos, V. (2008). Power laws in software. *ACM Trans. Softw. Eng. Methodol.*, 18(1). ISSN 1049-331X.

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

Martin, R. (1994). Oo design quality metrics. *An analysis of dependencies*, 12(1):151--170.

Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA. ISBN 0135974445.

Massacci, F., Neuhaus, S., and Nguyen, V. (2011). After-life vulnerabilities: A study on firefox evolution, its vulnerabilities, and fixes. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6542 LNCS:195–208.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*, (4):308--320.

Mens, T., Fernandez-Ramil, J., Fernandez-Ramil, J., and Degrandsart, S. (2008). The evolution of eclipse. In *ICSM*, pages 386–395. ISSN 1063-6773.

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

Merlo, E., Dagenais, M., Bachand, P., Sormani, J., Gradara, S., and Antoniol, G. (2002). Investigating large software system evolution: the linux kernel. In *Proceedings 26th Annual International Computer Software and Applications*, pages 421--426. IEEE.

Meyer, B. (1997). *Object-oriented software construction*, volume 2. Prentice hall Englewood Cliffs.

Miles, J. (2014). *R Squared, Adjusted R Squared*. American Cancer Society.

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

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

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

Mitchell, A. and Power, J. F. (2005). Using object-level run-time metrics to study coupling between objects. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 1456--1462.

Mitchell, Á. and Power, J. F. (2006). 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.

Mitzenmacher, M. (2004). A brief history of generative models for power law and lognormal distributions. *Internet mathematics*, 1(2):226--251.

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

Munson, J. C. and Khoshgoftaar, T. M. (1992). Measuring dynamic program complexity. *IEEE software*, 9(6):48--55.

Munson, J. C. and Khoshgoftaar, T. M. (1996). *Software Metrics for Reliability Assessment*, page 493–529. McGraw-Hill, Inc., USA.

Myers, G. (1975). *Reliable Software Through Composite Design*. Petrocelli/Charter. ISBN 0884052842.

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

Newman, M. E. (2003). The structure and function of complex networks. *SIAM review*, 45(2):167--256.

Nikora, A. P. and Munson, J. C. (2004). Developing fault predictors for evolving software systems. In *Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No. 03EX717)*, pages 338--350. IEEE.

Nosek, J. T. and Palvia, P. (1990). Software maintenance management: Changes in the last decade. *Journal of Software Maintenance: Research and Practice*, 2(3):157–174.

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

Olbrich, S. M., Cruzes, D. S., and Sjøberg, D. I. (2010). Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *2010 IEEE International Conference on Software Maintenance*, pages 1--10. IEEE.

Ostrand, T. J. and Weyuker, E. J. (2002). The distribution of faults in a large industrial software system. *ACM SIGSOFT Software Engineering Notes*, 27(4):55--64.

Pan, W., Li, B., Ma, Y., and Liu, J. (2011). Multi-granularity evolution analysis of software using complex network theory. *Journal of Systems Science and Complexity*, 24(6):1068--1082.

Pearl, J. (2014). *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Elsevier.

Pinzger, M., Gall, H., Fischer, M., and Lanza, M. (2005). Visualizing multiple evolution metrics. pages 67–75.

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

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

Rani, A. and Chhabra, J. (2017). Evolution of code smells over multiple versions of softwares: An empirical investigation. volume 2017-January, pages 1093–1098.

Ratzinger, J., Fischer, M., and Gall, H. (2005). Evolens: Lens-view visualizations of evolution data. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 103--112. IEEE.

Ratzinger, J., Pinzger, M., and Gall, H. (2007). Eq-mine: Predicting short-term defects for software evolution. volume 4422 LNCS, pages 12 – 26, Braga, Portugal. ISSN 03029743.

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

Rufiange, S. and Fuhrman, C. P. (2014). Visualizing protected variations in evolving software designs. *Journal of Systems and Software*, 88:231--249.

Rufiange, S. and Melancon, G. (2014). Animatrix: A matrix-based visualization of software evolution. pages 137–146.

Saha, R. K., Roy, C. K., Schneider, K. A., and Perry, D. E. (2013). Understanding the evolution of type-3 clones: an exploratory study. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 139--148. IEEE Press.

Sangwan, R., Vercellone-Smith, P., and Neill, C. (2010). Use of a multidimensional approach to study the evolution of software complexity. *Innovations in Systems and Software Engineering*, 6(4):299–310.

Savić, M., Ivanović, M., and Radovanović, M. (2011). Characteristics of class collaboration networks in large java software projects. *Information Technology and Control*, 40(1):48--58.

Schaeffer, M. (2009). *Adobe Flash CS4 professional how-tos: 100 essential techniques*. Pearson Education.

Servant, F. and Jones, J. A. (2012). History slicing: Assisting code-evolution tasks. pages 1–11, Cary, NC, United states.

Shatnawi, R. and Li, W. (2008). 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.

Singh, G. and Ahmed, M. (2017). Effect of coupling on change in open source java systems.

Smith, N., Capiluppi, A., and Ramil, J. F. (2004). Qualitative analysis and simulation of open source software evolution. In *Proc. 5th Intern. Workshop Software Process Simulation and Modeling*.

Smith, N., Capiluppi, A., and Ramil, J. F. (2005). A study of open source software evolution data using qualitative simulation. *Software Process: Improvement and Practice*, 10(3):287--300.

Smith, N., Capiluppi, A., and Ramil, J. F. (2006). Agent-based simulation of open source evolution. *Software Process Improvement and Practice*, 11(4):423–434.

Sommerville, I. (2012). *Software Engineering*. Pearson, 9th edition.

Sousa, B. L., Ferreira, M. M., Ferreira, K. A. M., and Bigonha, M. A. S. (2019). 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. Association for Computing Machinery.

Spinellis, D. and Avgeriou, P. C. (2019). Evolution of the unix system architecture: An exploratory case study. *IEEE Transactions on Software Engineering*, pages 1–31. ISSN 0098-5589.

Stewart, K., Darcy, D., and Daniel, S. (2006). Opportunities and challenges applying functional data analysis to the study of open source software evolution. *Statistical Science*, 21(2):167–178.

Stopford, B. and Counsell, S. (2008). A framework for the simulation of structural software evolution. *ACM Trans. Model. Comput. Simul.*, 18:17:1–17:36. ISSN 1049-3301.

Tahvildari, L., Gregory, R., and Kontogiannis, K. (1999). An approach for measuring software evolution using source code features. pages 10–17, Takamatsu, Japan.

Telea, A. and Auber, D. (2008). Code flows: Visualizing structural evolution of source code. *Computer Graphics Forum*, 27(3):831 – 838. ISSN 01677055.

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. (2010). 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.

Terceiro, A., Chavez, C., Babar, M. A., Lundell, B., and van der Linden, F. (2009). Structural complexity evolution in free software projects: A case study. In *QACOS-OSSPL 2009: Proceedings of the Joint Workshop on Quality and Architectural Concerns in Open Source Software (QACOS) and Open Source Software and Product Lines (OSSPL).*

Terceiro, A., MendonÃ§a, M., Chavez, C., and Cruzes, D. (2012). Understanding structural complexity evolution: A quantitative analysis. pages 85–94.

Thomas, L., Schach, S. R., Heller, G. Z., and Offutt, J. (2009). Impact of release intervals on empirical research into software evolution, with application to the maintainability of linux. *IET software*, 3(1):58--66.

Thomas, S., Adams, B., Hassan, A., and Blostein, D. (2014). Studying software evolution using topic models. *Science of Computer Programming*, 80(PART B):457–479.

Trindade, R., Orfanó, T., Ferreira, K., and Wanner, E. (2017). The dance of classes - a stochastic model for software structure evolution. In *WETSoM*, pages 22--28. ISSN 2327-0969.

Tu, Q. and Godfrey, M. W. (2002). An integrated approach for studying architectural evolution. In *Proceedings 10th International Workshop on Program Comprehension*, pages 127--136. IEEE.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., and Poshyvanyk, D. (2015). 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.

Tupper, C. D. (2011). Object and object/relational databases. In Tupper, C. D., editor, *Data Architecture*, pages 369--383. Morgan Kaufmann, Boston.

Turnu, I., Concas, G., Marchesi, M., Pinna, S., and Tonelli, R. (2011). A modified yule process to model the evolution of some object-oriented system properties. *Information Sciences*, 181(4):883–902.

Vasa, R., Lumpe, M., Branch, P., and Nierstrasz, O. (2009). Comparative analysis of evolving software systems using the gini coefficient. In *2009 IEEE International Conference on Software Maintenance*, pages 179--188. IEEE.

Vasa, R., Lumpe, M., and Jones, A. (2010). Helix - Software Evolution Data Set. `http://www.ict.swin.edu.au/research/projects/helix`.

Voinea, L., Telea, A., and Van Wijk, J. J. (2005). Cvsscan: visualization of code evolution. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 47--56. ACM.

Wang, L., Wang, Y., and Zhao, Y. (2014). Mechanism of asymmetric software structures: A complex network perspective from behaviors of new nodes. *Physica A: Statistical Mechanics and its Applications*, 413:162--172.

Wang, L., Wang, Z., Yang, C., and Zhang, L. (2012). Evolution and stability of linux kernels based on complex networks. *Science China Information Sciences*, 55(9):1972–1982.

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

Wang, L., Yu, P., Wang, Z., Yang, C., and Ye, Q. (2013). On the evolution of linux kernels: A complex network perspective. *Journal of software: Evolution and Process*, 25(5):439–458.

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

Weiser, M. (1984). Program slicing. *IEEE Transactions on software engineering*, (4):352--357.

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

Woodside, C. M. (1979). A mathematical model for the evolution of software. *Journal of Systems and Software*, 1:337--345.

Wu, J., Holt, R. C., and Hassan, A. E. (2004a). Exploring software evolution using spectrographs. In *11th Working Conference on Reverse Engineering*, pages 80--89. IEEE.

Wu, J., Spitzer, C. W., Hassan, A. E., and Holt, R. C. (2004b). Evolution spectrographs: Visualizing punctuated change in software evolution. In *Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.*, pages 57--66. IEEE.

Xie, G., Chen, J., and Neamtiu, I. (2009). Towards a better understanding of software evolution: An empirical study on open source software. In *ICSM*, pages 51--60. ISSN 1063-6773.

Xing, Z. and Stroulia, E. (2004). Understanding class evolution in object-oriented software. volume 12, pages 34–43.

Xing, Z. and Stroulia, E. (2005). Analyzing the evolutionary history of the logical design of object-oriented software. *IEEE Transactions on Software Engineering*, 31(10):850–868. ISSN 00985589.

Yacoub, S. M., Ammar, H. H., and Robinson, T. (1999). Dynamic metrics for object oriented designs. In *Proceedings Sixth International Software Metrics Symposium (Cat. No. PR00403)*, pages 50--61. IEEE.

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

Yu, L. (2007). Understanding component co-evolution with a study on linux. *Empirical Software Engineering*, 12(2):123–141.

Yu, L. and Ramaswamy, S. (2009). Measuring the evolutionary stability of software systems: Case studies of linux and freebsd. *IET Software*, 3(1):26–36.

Zhang, J., Sagar, S., and Shihab, E. (2013). 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.

Zheng, X., Zeng, D., Li, H., and Wang, F. (2008). Analyzing open-source software systems as complex networks. *Physica A: Statistical Mechanics and its Applications*, 387(24):6190--6200.

Zhou, Y., Wursch, M., Giger, E., Gall, H., and Lu, J. (2008). A bayesian network based approach for change coupling prediction. pages 27–36, Antwerp, Belgium. ISSN 10951350.

Zoubek, F., Langer, P., and Mayerhofer, T. (2018). Visualizations of evolving graphical models in the context of model review. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '18, pages 381--391, New York, NY, USA.

# Appendix A

# Software Evolution Metrics

This appendix provides an overview of the 72 software evolution metrics identified in this SLR. Tables A.1 and A.2 present a summary of the full set of metrics by includes the metrics name, metrics acronym, and papers references that mentioned them.

Table A.1: Part 1 - Overview of the software evolution metrics found in the SLR.

| ID | Name | Acronym | [Ref.] |
|---|---|---|---|
| 1 | Afferent coupling | CA | [Alenezi and Zarour, 2015] |
| 2 | Assignment | ASGMT | [Thomas et al., 2014] |
| 3 | Changed files | CF | [Terceiro et al., 2012] |
| 4 | Cohesion among methods of class | CAM | [Alenezi and Zarour, 2015] |
| 5 | Contribution period before change | CPBF | [Terceiro et al., 2012] |
| 6 | Coupling between methods | CBO | [Alenezi and Zarour, 2015] |
| 7 | CplXLCoh | CPLXLCOH | [Stewart et al., 2006; Darcy et al., 2010] |
| 8 | Cyclomatic complexity | CC | [Capiluppi and Ramil, 2004; Al-Ajlan, 2009] |
| 9 | Decrease in structural complexity | ΔSCd | [Terceiro et al., 2012] |
| 10 | Depth of folder tree | DFP | [Capiluppi and Ramil, 2004] |
| 11 | Depth of inheritance tree | DIT | [Nasseri et al., 2008] |
| 12 | Efferent coupling | CE | [Alenezi and Zarour, 2015] |
| 13 | Fan-in | FAN-IN | [Singh and Ahmed, 2017] |
| 14 | Fan-out | FAN-OUT | [Vasa et al., 2009; Singh and Ahmed, 2017] |
| 15 | Halstead's length | HLENG | [Herraiz et al., 2007] |
| 16 | Halstead's level | HLEVE | [Herraiz et al., 2007] |
| 17 | Halstead's volume | HVOLU | [Capiluppi and Ramil, 2004; Herraiz et al., 2007] |
| 18 | Halstead's mental discriminations | HMD | [Herraiz et al., 2007] |
| 19 | Implementation changes | IMC | [Singh and Ahmed, 2017] |
| 20 | Improved variation of the LCOM | LCOM3 | [Alenezi and Zarour, 2015] |
| 21 | In-degree count | IDC | [Vasa et al., 2009] |
| 22 | Increase in structural complexity | ΔSCi | [Terceiro et al., 2012] |
| 23 | Lack of cohesion in methods | LCOM | [Alenezi and Zarour, 2015] |
| 24 | Lines of code | LOC | [Godfrey and Tu, 2000; Capiluppi et al., 2004a,b; Robles et al., 2005; Herraiz et al., 2006; Izurieta and Bieman, 2006; Herraiz et al., 2007; Koch, 2007; Gonzalez-Barahona et al., 2009; Thomas et al., 2009; Darcy et al., 2010; Grigorio et al., 2015; Hatton et al., 2017] |
| 25 | Load instruction count | LIC | [Vasa et al., 2009] |
| 26 | Max function complexity | MaxFC | [Antinyan et al., 2013] |
| 27 | McCabe's complexity | VG | [Capiluppi et al., 2007; Herraiz et al., 2007; Antinyan et al., 2013; Alenezi and Zarour, 2015; Grigorio et al., 2015] |
| 28 | Method lines of code | MLOC | [Al-Ajlan, 2009] |
| 29 | Number of attributes | NOA | [Vasa et al., 2009] |
| 30 | Number of blank lines | BLKL | [Herraiz et al., 2007] |
| 31 | Number of children | NOC | [Nasseri et al., 2008] |
| 32 | Number of comment lines | CMTL | [Herraiz et al., 2007] |
| 33 | Number of commits | NOCOM | [Terceiro et al., 2012] |
| 34 | Number of designers | NODSR | [Antinyan et al., 2013] |
| 35 | Number of directories | NODIR | [Izurieta and Bieman, 2006] |
| 36 | Number of files | NOFL | [Capiluppi et al., 2004b; Herraiz et al., 2006; Izurieta and Bieman, 2006; Capiluppi et al., 2007] |

Table A.2: Part 2 - Overview of the software evolution metrics found in the SLR.

| ID | Name | Acronym | [Ref.] |
|---|---|---|---|
| 37 | Number of files added | NOFLA | [Capiluppi and Ramil, 2004] |
| 38 | Number of files deleted | NOFLD | [Capiluppi and Ramil, 2004] |
| 39 | Number of files handled | NOFLH | [Capiluppi et al., 2007] |
| 40 | Number of files modified | NOFLM | [Capiluppi and Ramil, 2004] |
| 41 | Number of files per level | NOFLPL | [Capiluppi et al., 2004b] |
| 42 | Number of files with decreasing McCabe index | NOFDVGI | [Capiluppi and Ramil, 2004] |
| 43 | Number of folders | NOFLR | [Capiluppi et al., 2004b] |
| 44 | Number of function returns | RETUR | [Herraiz et al., 2007] |
| 45 | Number of functions | FUNC | [Herraiz et al., 2007] |
| 46 | Number of instances of common coupling | NOICC | [Thomas et al., 2009] |
| 47 | Number of methods | NOM | [Vasa et al., 2009] |
| 48 | Number of non-commented lines | NONCL | [Antinyan et al., 2013] |
| 49 | Number of packages | NOP | [Gonzalez-Barahona et al., 2009] |
| 50 | Number of post-deployment failures | NOPDF | [Krishnan et al., 2011a] |
| 51 | Number of revisions per file | NORPF | [Antinyan et al., 2013] |
| 52 | Number of statements | NOSTM | [Al-Ajlan, 2009] |
| 53 | Out-degree count | ODC | [Vasa et al., 2009] |
| 54 | Public interface changes | PIC | [Singh and Ahmed, 2017] |
| 55 | Public method count | PMC | [Vasa et al., 2009] |
| 56 | Relative level of complexity control work | RLCPCW | [Capiluppi et al., 2007] |
| 57 | Response for a class | RFC | [Alenezi and Zarour, 2015] |
| 58 | Reuse ratio | RR | [Nasseri et al., 2008] |
| 59 | Scattering | SCAT | [Thomas et al., 2014] |
| 60 | Size of packages | SOP | [Gonzalez-Barahona et al., 2009] |
| 61 | Source code distace | SCDIST | [Yu and Ramaswamy, 2009] |
| 62 | Specialization ratio | SR | [Nasseri et al., 2008] |
| 63 | Store instruction count | SIC | [Vasa et al., 2009] |
| 64 | Structural complexity variation | $\Delta$SC | [Terceiro et al., 2012] |
| 65 | Structure distance | STRUCDIST | [Yu and Ramaswamy, 2009] |
| 66 | Total size in KB | SIZKB | [Capiluppi et al., 2004a,b; Izurieta and Bieman, 2006] |
| 67 | Type construction count | TYCC | [Vasa et al., 2009] |
| 68 | Variation in size of the project | $\Delta$LOC | [Capiluppi et al., 2007; Terceiro et al., 2012] |
| 69 | Weight | WGT | [Thomas et al., 2014] |
| 70 | Weighted methods per class | WMC | [Vasa et al., 2009; Alenezi and Zarour, 2015] |
| 71 | Width of folder tree | WFT | [Capiluppi and Ramil, 2004] |
| 72 | Width of level | WL | [Capiluppi and Ramil, 2004] |