# PREDIZENDO DEFEITOS DE SOFTWARE COM TESTES DE CAUSALIDADE

CÉSAR FRANCISCO DE MOURA COUTO

# PREDIZENDO DEFEITOS DE SOFTWARE COM TESTES DE CAUSALIDADE

Tese apresentada 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.

Orientador: Roberto da Silva Bigonha
Coorientador: Marco Túlio de Oliveira Valente, Nicolas Anquetil

Belo Horizonte

Dezembro de 2013

CÉSAR FRANCISCO DE MOURA COUTO

# PREDICTING SOFTWARE DEFECTS WITH CAUSALITY TESTS

Thesis presented to the Graduate Program in Ciência da Computação of the Universidade Federal de Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Ciência da Computação.

Advisor: Roberto da Silva Bigonha
Co-Advisor: Marco Túlio de Oliveira Valente,
Nicolas Anquetil

Belo Horizonte
December 2013

Couto, César Francisco de Moura

C871p Predicting Software Defects with Causality Tests / César Francisco de Moura Couto. — Belo Horizonte, 2013
xxiii, 106 f. : il. ; 29cm

Tese (doutorado) — Universidade Federal de Minas Gerais
Orientador: Roberto da Silva Bigonha

1. Defect Prediction; Causality; Software Metrics; Granger Test. I. Título.

CDU 519.6*32(043)

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

FOLHA DE APROVAÇÃO

Predicting software defects with causality tests

**CÉSAR FRANCISCO DE MOURA COUTO**

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

Prof. Roberto da Silva Bigonha - Orientador
Departamento de Ciência da Computação - UFMG

[illegible]

Prof. Marco Túlio de Oliveira Valente - Coorientador
Departamento de Ciência da Computação - UFMG

[illegible]

Prof. Nicolas Anquetil - Coorientador
Universidade de Lille-1 - França

[illegible]

Prof. Dalton Dario Serey Guerrero
Departamento de Sistemas e Computação - UFCG

[illegible]

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

[illegible]

Prof. Paulo César Masiero
Departamento de Sistemas de Computação - ICMC

[illegible]

Prof. Renato Martins Assunção
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 10 de dezembro de 2013.

*This thesis is dedicated to my wife Cinthia, who has always supported me.*

# Acknowledgments

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

I thank God to provide me the discipline and persistence to reach a Ph.D. degree.

I thank my wife Cinthia, who has had patience in difficult moments and has always been at my side.

I thank my whole family—especially my father Célio, my mother Cecília, my brothers Mila, Carol, and Celinho, and my mother-in-law Dalva—for having always supported me.

I thank my advisors M. T. Valente and R. S. Bigonha for the lessons, attention, availability, and patience.

I thank my co-advisor N. Anquetil for giving me the opportunity to work under his supervision in France.

I thank the members of the ASERG research group—especially R. Terra and C. Maffort—for the friendship and technical collaboration.

I thank the professor N. Vieira for the lessons during my teaching activities.

I thank the Department of Computer Science at UFMG, for the opportunity to participate its PhD program.

I would like to express my gratitude to the member of my thesis defense committee—E. Figueiredo (UFMG), R. Assunção (UFMG), D. Serey (UFCG), and P. C. Masiero (USP).

# Resumo

Predição de defeitos é uma área de pesquisa em engenharia de software que objetiva identificar os componentes de um sistema de software que são mais prováveis de apresentar defeitos. Apesar do grande investimento em pesquisa objetivando identificar uma maneira efetiva para predizer defeitos em sistemas de software, ainda não existe uma solução amplamente utilizada para este problema. As atuais abordagens para predição de defeitos apresentam pelo menos dois problemas principais. Primeiro, a maioria das abordagens não considera a idéia de causalidade entre métricas de software e defeitos. Mais especificamente, os estudos realizados para avaliar as técnicas de predição de defeitos não investigam em profundidade se as relações descobertas indicam relações de causa e efeito ou se são coincidências estatísticas. O segundo problema diz respeito a saída dos atuais modelos de predição de defeitos. Tipicamente, a maioria dos modelos indica o número ou a existência de defeitos em um componente no futuro. Claramente, a disponibilidade desta informação é importante para promover a qualidade de software. Entretanto, predizer defeitos logo que eles são introduzidos no código é mais útil para mantenedores que simplesmente sinalizar futuras ocorrências de defeitos.

Para resolver estas questões, nós propomos uma abordagem para predição de defeitos centrada em evidências mais robustas no sentido de causalidade entre métricas de código fonte (como preditor) e a ocorrência de defeitos. Mais especificamente, nós usamos um teste de hipótese estatístico proposto por Clive Granger (Teste de Causalidade de Granger) para avaliar se variações passadas nos valores de métricas de código fonte podem ser usados para predizer mudanças em séries temporais de defeitos. Nossa abordagem ativa alarmes quando mudanças realizadas no código fonte de um sistema alvo são prováveis de produzir defeitos. Nós avaliamos nossa abordagem em várias fases da vida de quatro sistemas implementados em Java. Nós alcançamos um precisão média maior do que 50% em três dos quatro sistemas avaliados. Além disso, ao comparar nossa abordagem com abordagens que não são baseadas em testes de causalidade, nossa abordagem alcançou uma precisão melhor.

# Abstract

Defect prediction is a central area of research in software engineering that aims to identify the components of a software system that are more likely to present defects. Despite the large investment in research aiming to identify an effective way to predict defects in software systems, there is still no widely used solution to this problem. Current defect prediction approaches present at least two main problems in the current defect prediction approaches. First, most approaches do not consider the idea of causality between software metrics and defects. More specifically, the studies performed to evaluate defect prediction techniques do not investigate in-depth whether the discovered relationships indicate cause-effect relations or whether they are statistical coincidences. The second problem concerns the output of the current defect prediction models. Typically, most indicate the number or the existence of defects in a component in the future. Clearly, the availability of this information is important to foster software quality. However, predicting defects as soon as they are introduced in the code is more useful to maintainers than simply signaling the future occurrences of defects.

To tackle these questions, in this thesis we propose a defect prediction approach centered on more robust evidences towards causality between source code metrics (as predictors) and the occurrence of defects. More specifically, we rely on a statistical hypothesis test proposed by Clive Granger to evaluate whether past variations in source code metrics values can be used to forecast changes in time series of defects. The Granger Causality Test was originally proposed to evaluate causality between time series of economic data. Our approach triggers alarms whenever changes made to the source code of a target system are likely to present defects. We evaluated our approach in several life stages of four Java-based systems. We reached an average precision greater than 50% in three out of the four systems we evaluated. Moreover, by comparing our approach with baselines that are not based on causality tests, it achieved a better precision.

# List of Figures

# List of Tables

# Contents

# Chapter 1

# Introduction

In this chapter, we start by presenting our motivation (Section 1.1) and current research challenges on defect prediction (Section 1.2). Next, we present our thesis statement and an overview of our approach for predicting defects in software systems (Section 1.3). Finally, we present the outline of this thesis (Section 1.4).

## 1.1 Motivation

The primary goal of software engineering is to produce high quality software [Mey00, p. 3]. However, producing software that is fast, reliable, easy to use, readable, and well-structured is not a simple task. Software development is a complex and challenging task because it involves a number of technologies, practices, and methodologies, such as requirements specification, programming languages, databases, design patterns, platforms, frameworks, and processes.

On the other hand, software maintenance is as complex and challenging as software development. It involves tasks such as: (a) corrective maintenance that concerns fixing bugs; (b) adaptive maintenance that corresponds to adaptations in software to changes in its environment; (c) perfective maintenance that deals with improving or adding new features in the system's requirements; and (d) preventive maintenance that concerns activities aiming to increase the system's maintainability [LS80]. Software maintenance takes up most part of software cost [LS80, NP90, Mey00, Som09]. According to Sommerville, 50%–75% of the total software costs concern the maintenance tasks [Som09, Ch. 9]. In this total cost, 17% concerns corrective maintenance tasks, 18% corresponds to adaptive maintenance tasks, and 65% deals with perfective and preventive maintenance tasks.

The high cost of maintenance is influenced by various factors. Among these factors, we can mention the complexity of the problem domain, the turnover of developers, the lack of documentation, and the low quality of the source code under maintenance. Particularly, some studies investigated the impact of the low quality of the source code on the maintenance costs [LB85, Cor89, Vis10]. Lehman and Belady were among the first to observe that as a software system increases in size and complexity, its quality decreases and its maintenance becomes harder [LB85]. Corbi showed that 50%–60% of the time of a maintenance task is spent understanding the source code [Cor89]. According to Visser, low quality software systems are resistant to change, because the more complex, unstructured, or tangled is the source code, the higher is the time spent to perform a maintenance task [Vis10].

On the other hand, evaluating a software system in order to improve its overall quality is also a challenging task. Meyer has proposed a set of properties that can be used to evaluate software quality [Mey00, p. 3]. According to Meyer, software quality can be evaluated by external factors, i.e., those factors perceived by users, and internal factors, i.e., those factors only perceived by the development team (developers and maintainers). Among the external quality factors, we can mention properties such as efficiency, correctness, robustness, extensibility, reusability, and ease of use. On the other hand, the internal quality factors include properties such as coupling, cohesion, readability, modularity, separation of concerns, etc.

In recent decades, many metrics have been proposed to evaluate both internal and external software properties [Hum95, FP97, Kan02, LMD05, Pre10]. For example, internal quality factors can be measured by source code metrics, including properties such as coupling, cohesion, size, inheritance, complexity, and violations in recommended programming practices. On the other hand, the external quality can be measured for example by the number of bugs reported by the users and developers.[1] In summary, these metrics provide a quantitative indication of some properties of a software system. Therefore, designers, developers, and maintainers can rely on these metrics to evaluate and control the internal and external quality of a software system. Potentially, such quality control can prevent future problems, as the occurrence of bugs.

Particularly, the number of bugs is an important measure to evaluate the reliability (correctness and robustness as proposed by Meyer [Mey00]) of software systems. Reliability is a critical external factor because it can impair the success of the product ahead stakeholders and increase the costs of corrective maintenance tasks. Therefore, knowing in advance the chances that a system has to fail in the future is essential to

[1]In our terminology, we consider that bugs are failures in the observable behavior of systems. Bugs are caused by one or more errors in the source code, called defects [Sta90].

increase its reliability, and ultimately its quality. In this context, defect prediction is an important area of research in software engineering that aims to identify the components of a system that are more likely to fail [BBM96, SK03, NB05a, NBZ06, MPS08, Has09]. Clearly, the availability of this information is of central value to most software quality assurance procedures. For example, it allows quality managers to allocate more time and resources to test, redesign, and reimplement those components predicted as defect-prone [ZNZ08]. In general, the aforementioned quality assurance procedures can reduce the amount of corrective maintenance and therefore improve customer's satisfaction.

Figure 1.1 summarizes the typical scheme followed by the state-of-the-art in defect prediction approaches [HBB+12]. Typically, these approaches work by retrieving information on a system such as bug history (extracted from bug tracking platforms) and source code versions (extracted from version control platforms), computing software metrics, and after that building prediction models to identify the components of a software that are defect-prone. More specifically, current defect prediction approaches aim to determine the number of defects (during a period of analysis) in a software component. Basically, the defects of a component are compared to other properties (measured by software metrics) to infer which properties typically correlate with defects. The ultimate goal is to construct a model that predicts the number or the existence of defects in a component in a future time frame.

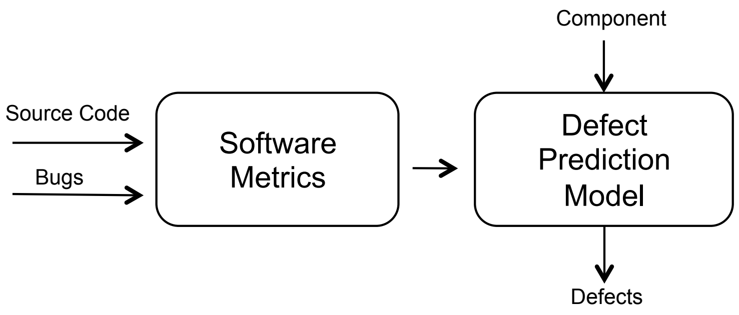


Figure 1.1: Overview of defect prediction approaches

## 1.2 Research Challenges on Defect Prediction

Due to its central relevance to software quality, various defect prediction techniques have been proposed. A recent systematic literature review identified 208 defect prediction studies published from January 2000 to December 2010 in major software engineering journals and conferences [HBB+12]. Basically, these studies differ in terms of the software metrics used for prediction, the modeling technique, the granularity of the independent variable, and the validation technique [HBB+12]. Figure 1.2 summarizes the variables and techniques involved in current defect prediction approaches. The independent variables include source code metrics, change metrics, warnings issued by bug finding tools, and violations in recommended programming practices (or code smells). The modeling techniques include linear regression, logistic regression, naïve bayes, neural networks, decision tree. The granularity of the prediction can be at the file/class/method level or module/package level. The validation phase can rely on classification and ranking techniques.

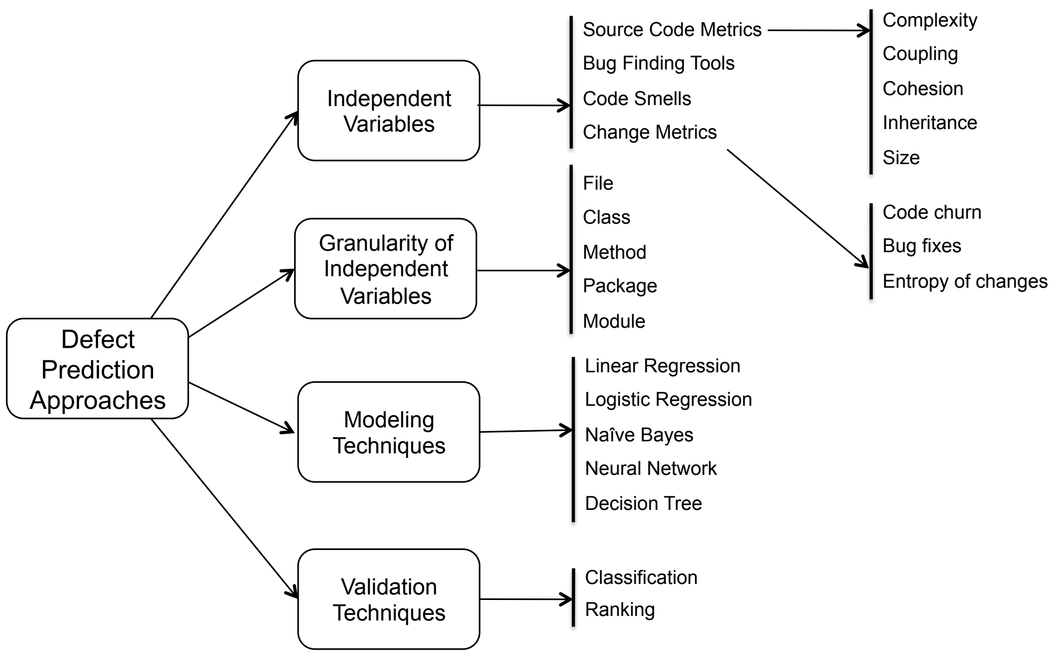


Figure 1.2: Variables and techniques used by defect prediction approaches

Despite the large investment in research aiming to identify an effective way to predict defects in software systems, we still lack a off-the-shelf solution to this problem. After a careful review of the literature, we identified at least two main problems in current defect prediction approaches. First, most approaches described in the literature do not consider the idea of causality between software metrics and defects. In other words, most studies do not investigate in-depth whether the discovered relationships indicate cause-effect relations or whether they are statistical coincidences. For example, the most common modeling technique used by defect predictors is linear or logistic regressions [BBM96, SK03, NB05a, GFS05, DLR10]. However, linear and logistic regressions do not imply causality [Ful94]. As a consequence, whenever we identify a linear or logistic relationship between two variables, we can not automatically conclude that one of the variables is the cause of (or directly affects) the other. More specifically, it is well known that regression models cannot filter out spurious correlations [Ful94].

The second problem we identified concerns the output of the current defect prediction models. As described in Section 1.1, typically the output of such prediction models indicate the number or the existence of defects in a component in the future. Clearly, the availability of this information is important to foster software quality. However, predicting defects as soon as they are introduced in the source code, e.g., identifying the changes to a class that are more likely to generate defects, is more useful to the maintainer than simply signaling the future occurrences of defects (since defects are expected anyway in most real-world software components).

## 1.3 Thesis Statement

Our thesis statement is as follows:

> *Reliable and precise defect prediction techniques play a pivotal role on software quality assessment and improvement. However, the state-of-the-art defect prediction approaches are centered on techniques that have not been designed to capture temporal cause-effect relations. Therefore, the main goal of this work is to propose a new defect prediction model centered on causal relations over time series of source code metrics and software defects. The proposed model triggers defect alarms whenever changes made to the source code of a target system have a high chance of producing defects.*

In this thesis, we propose a defect prediction approach centered on more robust evidences towards causality between source code metrics (as predictors) and the occurrence of defects. More specifically, we rely on a statistical hypothesis test proposed by Clive Granger to evaluate whether past changes to a given source code metrics time series can be used to forecast changes in software defects time series [Gra69, Gra81]. The Granger Causality Test was originally proposed to evaluate causality between time series of economic data (e.g., to show whether changes in oil prices cause recession). Although extensively used by econometricians, the test was also applied in bioinformatics (to identify gene regulatory relationships [MC07]) and recently in software maintenance (to detect change couplings spread over an interval of time [CCPC10]). Unlike common approaches in the literature, the model we propose does not aim to predict the number or the existence of defects in a source code class in a future time frame [BBM96, BWIL99, SK03, GFS05, DLR10]. Instead, the central goal of our approach is to predict defects as soon as they are introduced in the source code. More specifically, we aim to identify the changes to a class that are more likely to generate defects. For this purpose, our approach relies on input from the Granger Test to trigger alarms as soon as changes that are likely to introduce defects in a class are made. Therefore, we claim that our model contributes directly to improve software quality.

Figure 1.3 details our approach for defect prediction. In a first step, we apply the Granger Causality Test to infer possible causalities between historical values of source code metrics and the number of defects in each class of the system under analysis. In this first step, we also calculate a threshold for variations in the values of source code metrics that in the past Granger-caused defects in such classes. For example, suppose that a Granger-causality is found between changes in the size of a given class in terms of lines of code (LOC) and the number of defects in this class. Considering previous changes in this specific class, we can establish for example that changes adding more than 50 lines of code are more likely to introduce defects (more details on how such thresholds are calculated are presented in Chapter 6). Using these thresholds and the Granger results calculated in the previous step, a defect predictor analyzes each change made to a class and triggers defect alarms when similar changes in the past Granger-caused defects.

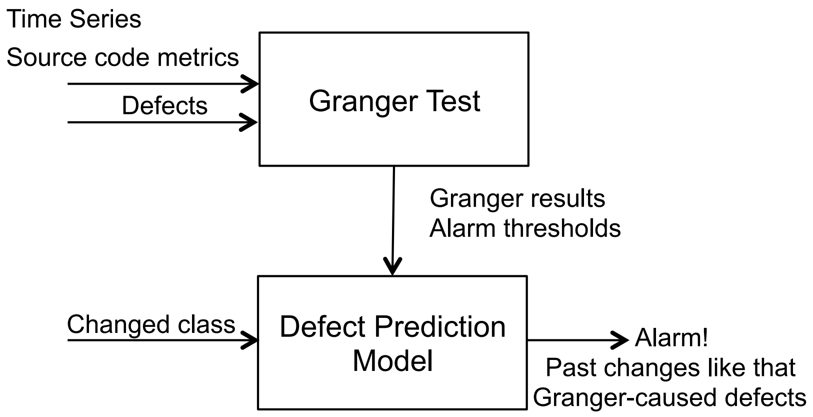


Figure 1.3: Proposed approach to predict defects

To develop the proposed thesis, we performed the following tasks:

1. We conducted a first study on the effectiveness of the warnings issued by bug finding tools. The main goal was to evaluate whether the warnings reported by such tools can be used as an independent variable in the construction of defect prediction models. For this purpose, we evaluated the effectiveness of the FindBugs tool—an important tool widely used by the Java developer community [HP04]. This study resulted in a work that received a best paper award in the Brazilian

Conference on Software Quality [ACSV10]. Later, this study was extended and published in the Software Quality Journal [CASV13]. The methodology, dataset, results, and lessons learned after this first study are described in Chapter 2.

2. We extended a dataset made public by D'Ambros et al. to evaluate defect prediction techniques [DLR10]. Basically, we extended this dataset: (a) by extracting again all source code versions considered in the dataset and recalculating the source code metrics; (b) by almost doubling the number of source code versions included in the original dataset, and (c) by introducing the time series of defects. This dataset is part of the COMETS dataset, which generated a communication in Software Engineering Notes [CMGV13]. Chapter 4 describes our extension to D'Ambros et al. dataset.

3. We conducted a second study to investigate the feasibility of applying Granger to detect causal relationships between time series of source code metrics and defects. Particularly, our goal was to evaluate whether there are causal relationships between source code metrics and defects in the classes of object-oriented systems. This study resulted in a working paper in the Brazilian Conference on Software Quality [CVB11] and full paper in the European Conference on Software Maintenance and Reengineering [CSV+12]. The methodology, results, and lessons learned after this second study are described in Chapter 5.

4. We developed and evaluated an approach for predicting defects using causality tests. More specifically, we leveraged the experience and knowledge gained after the studies described in the previous items to propose and validate a model that triggers alarms whenever changes made to the source code of a target system have a high chance of producing defects. Chapter 6 describes the steps we followed to construct and evaluate this model.

5. We designed and implemented a prototype tool for the visual exploration and analysis of bugs, called BugMaps. The implementation of this tool resulted in a paper presented in the tool demonstration track in the European Conference on Software Maintenance and Reengineering [HCA+12], which received the highest scores by the track reviewers. In addition, we extended this tool to support the visualization of causality relations between source code metrics and bugs, and we called this extension BugMaps-Granger. The implementation of this tool resulted in a paper presented in the tools session of the Brazilian Conference on Software [CPV+13]. The BugMaps-Granger tool received the best tool award of this conference.

## 1.4 Thesis Outline

This thesis is structured in the following chapters:

- Chapter 2 provides a general discussion on software quality and defect prediction and presents the software metrics commonly used to predict defects. This chapter also presents the state-of-the-art in defect prediction.
- Chapter 3 presents an overview on the Granger Causality Test and describes other techniques commonly used for predicting defects.
- Chapter 4 describes our dataset including time series of source code metrics and defects, extracted for four real world systems (Eclipse JDT Core, Eclipse PDE UI, Equinox, and Lucene).
- Chapter 5 describes a feasibility study designed to illustrate and to evaluate the application of Granger on defects prediction.
- Chapter 6 describes the defect prediction approach proposed in this work, as well its evaluation.
- Chapter 7 presents the BugMaps tool for the visual exploration and analysis of bugs, including the visualization of causal relations between source code metrics and bugs.
- Chapter 8 concludes this PhD thesis.

# Chapter 2

# Background

In this chapter, we start by providing a discussion about software quality and defect prediction (Section 2.1). Next, we present the software metrics commonly used by defect prediction approaches (Section 2.2). Finally, we present the state-of-the-art in defect prediction (Section 2.3) and provide a critical appraisal on current defect prediction approaches.

## 2.1 Software Quality and Defect Prediction

The primary goal of software engineering is to produce high quality software. Software quality is the degree to which a software meets its requirements specification [Sta90]. On the other hand, software quality assurance consists of a set of activities necessary to provide adequate confidence that a software conforms to its requirements specification [Sta90]. Among the software quality activities, we can mention code review, refactoring, testing, measuring the impact of changes, keeping records and reporting (bugs, improvements, and new features), configuration management, release management, product integration, etc.

To clarify how defect prediction approaches might be useful for ensuring software quality, suppose a scenario where a version of a given software will be released but the developer leader suspects that there are defects in some source code modules. To find these defects, the developer leader can rely on some resources responsible for ensuring software quality, such as code reviewers, senior developers, and testers. However, these resources are limited and represent a high cost to the software project. Therefore, the developer leader want to spend them in the most effective way, getting the best software quality and the lowest risk of defects. More specifically, he wants to spend the most quality assurance resources on those modules that need it most, i.e., those modules that have high chances of producing defects.

On the other hand, allocating quality assurance resources is not a simple task. If a module without defects is tested or reviewed over a long period, this may indicate a non-optimal allocation of resources. If a module with defects is not tested or reviewed enough, a defect can appear in the field (i.e., defects reported by final users) causing more serious problems, such as corrective maintenance and stakeholders dissatisfaction. Therefore, identifying defect-prone modules may help the development leader to decide where the software quality assurance resources must be allocated before releasing a new version of a target software. Particularly, the quality assurance resources can perform tasks such as source code inspection, unit testing, functional testing, etc. Such tasks can improve the reliability of the software under development, reducing the number of future corrective maintenance tasks and improving the customer satisfaction.

In summary, knowing in advance the chances that a system has to fail in the future is a desirable way to obtain a more correct and robust software. It is in this context that the studies on defect prediction are addressed. Various defect prediction approaches have been proposed in recent decades [BBM96, SK03, NB05a, NBZ06, MPS08, Has09]. These approaches analyze historical information about a target software—such as bug history and source code evolution—in search of indicators about the presence of future defects. For example, a simple defect prediction approach can rely on the number of past defects as an indicator of future defects, i.e., software modules that had many defects in the past might have more chances of future defects. Among other metrics used as indicators of defects, we can mention source code metrics, code change metrics, warnings issued by bug finding tools, design flaws, etc.

In the following sections, some important software metrics used by defect prediction approaches are presented and the state-of-the-art in defect prediction is reviewed. Finally, an evaluation of the current works on defect prediction is presented.

## 2.2 Software Metrics

A software metric is defined as "*quantitative measure of the degree to which a system, component or process possesses a given attribute*" [Sta90]. More specifically, it can be illustrated as a function that receives as input software data and returns as output a numerical value that represent the degree to which the software possesses a given quality attribute. Typically, software metrics provide insight that enable software engineers to adjust and control software products, projects and processes [Pre10]. Moreover, software metrics are usually classified into three categories: process metrics, project metrics, and product metrics [Kan02, Pre10], as described next:

- Process metrics: enable the organization to evaluate the process used to construct a software system. More specifically, a software process defines techniques, management methods, tools, people, and tasks related to the software development. Therefore, process metrics can be used to improve software development and maintenance practices. As examples, we can mention defects per KLOC or function point, change metrics, number files involved in bug fixing, etc.

- Project metrics: enable the organization to evaluate the progress of a software project. Basically, project metrics describe the project characteristics and execution. Number of developers, cost, schedule, and productivity are examples of project metrics.

- Product metrics: enable the software engineers to evaluate the internal properties of a software product. As examples of product metrics, we can mention size, complexity, coupling, cohesion, and inheritance.

Particularly, in this thesis, we focus on process and product metrics, since they have been used as independent variables in several defect prediction approaches. Among the product metrics already used to construct defect prediction models, we can mention source code metrics (including complexity, coupling, cohesion and size metrics) [BBM96, BWD$^{+}$00, SK03, GFS05, DLR10] and warnings reported by bug finding tools [NB05a, ZWN$^{+}$06, WAWS08, ASV11, CASV13]. On the other hand, process metrics such as code change metrics are used by other defect prediction models [GKMS00, NB05b, Has09, MPS08]. Finally, we are not aware of works that use project metrics to predict defects.

## 2.2.1 Source Code Metrics

Several defect prediction approaches reported in the literature are centered on source code metrics [BBM96, BWD$^{+}$00, SK03, GFS05, DLR10]. Typically, such approaches consider that the current design and structure of the software influence the presence of future defects. Basically, they do not analyze the version history of the system, but only its current codebase, by using a variety of source code metrics.

A set of metrics commonly used for the purpose of defect prediction is the CK metrics suite, which consists of the following metrics: Weighted Methods per Class

(WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling between Object Class (CBO), Response for a Class (RFC), and Lack of Cohesion in Methods (LCOM) [CK91, CK94]. These metrics are described next:

- WMC: represents the complexity of the class as measured by its methods. The calculation of the metric is given by the sum of the complexity of the methods in the class. However, the definition of complexity remains open. According to Chidamber and Kemerer, WMC is an indicator of how much time and effort are required to develop and maintain a given class.

- DIT: indicates the depth of a class in the inheritance tree, which is given by the length of the path from the class to the root of the tree. DIT is nowadays considered an indicator of design complexity.

- NOC: denotes the number of immediate subclasses of a class. This metric is an indicator of the importance that a class has in the system. If a class has a large number of children, it might for example require more tests.

- CBO: indicates the number of classes to which a certain class is coupled to. For Chidamber and Kemerer, a coupling between two classes exists when methods implemented in one class use methods or instance variables defined by other classes. This metric can be used to reveal design problems. For example, it is widely accepted that excessive coupling is harmful to modular design, because the more independent a class is, more easy is to reuse it in other applications.

- RFC: indicates the number of methods that can be called in response to a message received by a class, defined as the number of methods of the class plus the number of methods invoked by them. RFC is considered an indicator of coupling.

- LCOM: indicates the lack of cohesion between the methods in a class. Chidamber and Kemerer consider that cohesion between methods is defined by the use of common instance variables. LCOM is the number of method pairs that have no instance variables in common minus the number of method pairs with common instance variables. Therefore, the smaller the value of LCOM, the more cohesive is the class.

In summary, CK metrics cover different internal properties of software systems, such as complexity (WMC), coupling (CBO and RFC), inheritance (DIT and NOC), and cohesion (LCOM). Moreover, there are other object-oriented metrics used by defect prediction models [GFS05, DLR10, LMD05], as presented in Table 2.1. Among such metrics, we can mention lines of code (LOC), number of public methods (NOPM), FAN-IN, FAN-OUT, etc.

Table 2.1: Other object-oriented metrics

| **Metrics** | **Description** |
|---|---|
| FAN-IN | Number of classes that reference a given class |
| FAN-OUT | Number of classes referenced by a given class |
| NOA | Number of attributes |
| NOPA | Number of public attributes |
| NOPRA | Number of private attributes |
| NOAI | Number of attributes inherited |
| LOC | Number of lines of code |
| NOM | Number of methods |
| NOPM | Number of public methods |
| NOPRM | Number of private methods |
| NOMI | Number of methods inherited |

Finally, D'ambros et al. proposed a metric called entropy of source code to predict defects [DLR10, DLR12]. The basic idea consists in measuring the evolution of source code metrics through the variations of a metric over subsequent sample versions. The more spread the variations of the metric, the higher is the entropy. For example, suppose that the WMC of a system is 100, but only one class contributes to this result. For this source code metric, the entropy is low. On the other hand, if the WMC of a system is 100, but 10 classes have contributed equally to achieve this result, this means that the entropy is high.

### 2.2.2 Process Metrics

Process metrics are also used by studies on defect prediction [GKMS00, NB05b, MPS08, Has09, DLR12, KSA$^+$13]. Typically, defect prediction approaches based on process metrics consider that information extracted from version control systems—such as source code changes—can be used to predict defects. These approaches assume that code that changes a lot is more defect-prone than stable code. For example, Ball et al. introduced the concept of code churn as a measure of the "*amount of code change taking place within a software unit over time*" [NB05b]. As example of code churn, we can mention number of changed files and the sum of lines of code added, changed, and deleted between two versions.

Moser et al. introduced a catalog with a set of file-level change metrics (including number of revisions and refactorings, number of distinct authors, etc.) [MPS08]. Particularly, they proposed a metric called Bugfixes that denotes the number of times a file was involved in bug-fixing activities. To calculate this metric, the authors used pattern matching on the comments available in the commit operations. For determining whether a revision is a bug fix, the revision comment must match the string "%Fix%" and must not match the strings "%prefix%" and "%postfix%". On the other hand, Zimmermann et al relied on the convention that the IDs of the bugs are included in the comments of commits in bug fix operations [ZPZ07]. In other words, to be classified as a bug fix, the revision comments must include a reference to the bug ID. In this thesis, we adopted a similar strategy as described in Section 4.2.2.

Hassan introduced the metric entropy of changes, in order to measure the complexity of code changes [Has09]. Basically, this metric measures how distributed are the changes in the files of a target system during a time interval. The more spread is a change, the higher its complexity. The intuition is that a change that affects only a single file is simpler than one that affects several files. Therefore, this metric is an indicator of the complexity of code changes.

### 2.2.3 Warnings from Bug Finding Tools

Several bug finding tools have been proposed to detect software defects by means of static analysis techniques [NB05a, ZWN$^+$06, WAWS08, ASV11, CASV13]. Basically, defect prediction approaches based on bug finding tools try to infer relationships between the warnings issued by such tools and defects. In this section, we provide an overview of the FindBugs and PMD tools, which are commonly used in defect prediction approaches [HP04, Cop05].

FindBugs is an open-source tool that relies on static analysis to look for more than four hundred bug patterns in Java bytecode [HP04]. Bug patterns are coding idioms that are likely to represent errors and are classified into categories such as thread/synchronization correctness, malicious code, performance, etc. Bug patterns are also assigned high, medium, or low priorities. FindBugs internal architecture includes components for intraprocedural control and data flow analysis. These components are responsible for making a sequential search through the bytecode to detect bug patterns.

Figure 2.1 shows an example of a potential buggy code detected by FindBugs in the class `QueryParser` from the Apache Lucene system[1]. The catch block (line 1071) is empty, i.e., the exception is being ignored. Figure 2.2 shows the warning message generated by FindBugs after analyzing this code fragment. This warning indicates that exceptions should be handled or thrown out by the target method.

[1]http://lucene.apache.org/core/

```
final public Query Term(String field) throws ParseException {
       ...

  try {
       fms = Float.valueOf(fuzzySlop.image.substring(1))...
  } catch (Exception ignored) { }

       ...
}
```

Figure 2.1: Buggy code detected by FindBugs in the Apache Lucene

```
DE: Method might ignore exception (DE_MIGHT_IGNORE)
This method might ignore an exception.  In general, exceptions should be
handled or reported in some way, or they should be thrown out of the method.
```

Figure 2.2: FindBugs warning example

PMD is another open-source tool that uses static analysis to identify potential problems in Java source code and to check coding styles [Cop05]. For example, PMD rulesets are able to detect empty statements, dead code, duplicated code, inappropriate coupling, untrusted code, etc. Different from FindBugs, PMD requires the source code of the target program. Basically, PMD parses the source code in order to create an Abstract Syntax Tree (AST). To detect violations, PMD checks this AST against predefined rulesets.

Figure 2.3 shows an example of a code smell (i.e., a violation in recommended programming practices) detected by PMD in the class `LuceneMethods` from the Apache Lucene system. The method `invertDocument` has some nested `if` statements (lines 295 to 296). For this method, PMD generates the warning described in Figure 2.4, which recommends that the nested `if` should be combined in a single statement.

```
286: private void invertDocument(Document doc) {
        ...
295:      if (field.isIndexed()) {
296:        if (field.isTokenized()) {
297:          Reader reader;
        ...
338: }
```

Figure 2.3: Example of class with a warning generated by PMD

```
<violation
 beginline="296" endline="329"
 begincolumn="9" endcolumn="9"
 rule="CollapsibleIfStatements"
 ruleset="Basic"
 package="lucli"
 class="LuceneMethods"
 method="invertDocument"
 externalInfoUrl="http://pmd.sourceforge.net/rules/..."
 priority="3">

 These nested if statements could be combined

</violation>
```

Figure 2.4: PMD warning example (for the class showed in Figure 2.3)

## 2.3 Defect Prediction Approaches

A recent systematic literature review identified 208 defect prediction studies—including some of the works that will be presented in this section—published from January 2000 to December 2010 [HBB$^+$12]. The studies differ in terms of the software metrics used for prediction, the modeling technique, the granularity of the independent variable, and the validation technique. Typically, the independent variables are associated to source code metrics, change metrics, previous defects, warnings issued by bug finding tools, design flaws, etc. The modeling techniques vary with respect to linear regression, logistic regression, naïve bayes, neural networks, etc. The granularity of the prediction can be at the method level, file/class level, or module/package level. The validation are usually conducted using classification or ranking techniques.

This section presents the state-of-the-art in defect prediction and performs a critical appraisal of related work. The term "fault" is used interchangeably with the terms "defect" or "bug" to represent a static anomaly in the source code. On the other hand, a failure occurs when a unit of software is unable to perform its function [Sta90]. More specifically, the defect prediction approaches we discuss in this section are arranged in three groups: (a) source code metrics approaches; (b) process metrics approaches; and (c) bug finding tools approaches. Source code metrics approaches consider that the current design and structure of the software influence the presence of future defects. Furthermore, approaches based on process metrics consider that information extracted from version control platforms such as code changes influence the occurrence of defects. On the other hand, approaches based on bug finding tools try to infer relationships between the warnings issued by such tools and defects. Finally, we have a fourth group that includes a study on the application of causality tests in software maintenance.

### 2.3.1 Source Code Metrics Approaches

The source code properties as measured by CK metrics received considerable attention for the purposes of defect prediction, as can be observed in Table 2.2. Basili et al. were among the first to investigate the use of CK metrics as early predictors for fault-prone classes [BBM96]. In a study on eight medium-sized systems they used logistic regression to analyze the relationship between metrics and fault-prone classes. This study revealed that there is a correlation between CK metrics (with the exception of the LCOM metric) and such classes.

Briand et al. discovered that several design metrics from the CK suite were positively associated with fault-prone classes [BWIL99, BWD$^{+}$00]. More specifically, the frequency of method invocations (related to CBO and RFC) and the depth of inheritance hierarchies (related to DIT) are associated with fault-prone at the level of classes. Cartwright and Shepperd studied the inheritance measures derived from the CK suite (DIT and NOC) in an industrial object-oriented system implemented in C++ [CS00]. They concluded that both measures have an influence on the defect density of classes. Emam et al. used the CK metrics and Briand's coupling metrics [BDW99] to predict fault-prone classes in a commercial Java system [EMM01]. The results indicated that inheritance (DIT) is strongly associated with fault-prone classes.

Subramanyam and Krishnan investigated the relation between defects and CK metrics, such as CBO, WMC, and DIT [SK03]. In their study, they evaluated a single e-commerce system with modules implemented in C++ and Java. For modules in C++, they found that WMC, DIT, and CBO with DIT have a relevant impact on the number of defects. For the modules implemented in Java, only CBO with DIT has had an impact on defects. Gyimothy et al. performed an analysis similar to the study conducted by Basili et al. [BBM96] using CK metrics and lines of code (LOC) as predictors for fault-prone classes [GFS05]. They showed that CBO is among the best metrics for fault prediction and LOC is the second metric with the best results.

D'ambros et al. provided the original dataset with the historical values of the source code metrics that will be used in this thesis [DLR10]. By making this dataset publicly available, their goal was to establish a common benchmark for comparing bug prediction approaches. They relied on this dataset to evaluate a representative set of prediction approaches reported in the literature, including approaches based on CK metrics, the object-oriented metrics showed in Table 2.1, change metrics, bug fixes, and entropy of changes. The results indicated that CK metrics yielded a poor predictor with an unstable behavior among the analyzed systems.

Other studies on defect prediction also used CK metrics as independent variables

Table 2.2: Empirical studies on CK metrics and defect prediction

| Study | Indep. Variable | Model Technique | Results |
|---|---|---|---|
| [BBM96] | CK suite | Logistic Regression | WMC, DIT, RFC, CBO, and NOC contributed to predict faults. LCOM did not contribute to predict faults. |
| [BWIL99] | CBO, RFC, and LCOM | Logistic Regression | CBO, RFC, and LCOM were associated with fault-prone of classes. |
| [BWD$^+$00] | CK suite | Logistic Regression | WMC, CBO, DIT, RFC and NOC were associated with fault-prone of classes. LCOM was not associated with faults. |
| [CS00] | DIT and NOC | Linear Regression | DIT and NOC had influence on the defect density of classes. |
| [EMM01] | DIT and NOC | Logistic Regression | DIT is associated with fault-prone. |
| [SK03] | WMC, CBO, and DIT | Linear Regression | WMC, CBO, and DIT had a relevant impact on the number of defects. |
| [GFS05] | CK suite | Logistic Regression, Linear Regression, and others | CBO contributed to predict faults. |
| [DLR10] | CK suite | Linear Regression | CK suite yielded a poor predictor. |

in their models [NBZ06, HPH$^+$09]. However, such studies did not analyze the CK suite individually, but combined with other source code metrics. For example, Nagappan et al. conducted a study on five components of the Windows operating system in order to investigate the relationship between source code metrics and field defects [NBZ06]. They concluded that source code metrics indeed correlate with defects. However, they highlight that there is no single set of metrics that can predict defects in all the five Windows components. As a consequence of this finding, the authors suggest that software quality managers can never blindly trust on metrics, i.e., in order to use metrics as early bug predictors we must first validate them using the project's history [ZNZ08].

Later, the study of Nagappan et al. was replicated by Holschuh et al. using a large ERP system (SAP R3) [HPH$^+$09]. They confirmed the results obtained by Nagappan et al. in the this new system. However, both studies rely on linear regression models and correlation tests, which consider only an "immediate" relation between the independent and dependent variables. On the other hand, the dependency between bugs and source code metrics may not be immediate, i.e., usually there is a delay or lag in this dependency. In this thesis, we presented a new approach for predicting bugs that considers this lag.

### 2.3.2 Process Metrics Approaches

Basically, process metrics approaches use information extracted from version control systems, assuming that recently or frequently changed files are more subjected to defects. Table 2.3 shows some of the approaches in the literature that rely on process metrics to predict defect-prone modules. As we can observe, Graves et al. reported a study based on the fault history of the modules of a large telephone switching system [GKMS00]. They found that lines of code and other standard source code metrics are generally poor predictors of faults. On the other hand, they argued that process metrics—such as number of modifications, the age of a file, size of the modifications, etc.—are better predictors of future faults.

Nagappan and Ball analyzed the code churn between the releases of the Windows Server 2003 and Windows Server 2003-SP1 to predict the defect density in later release [NB05b]. They found that relative code churn (e.g., normalized by the number of lines of code) is able to predict system defect density with high levels of statistical significance. Moser et al. used data extracted from the version control system of the Eclipse project, such as source code metrics (including complexity, size, etc.) and change metrics (including code churn, files committed together, number of times a file was involved in bug-fixing, etc.) to predict a code unit either as defect free or defective [MPS08]. The results indicated that for the Eclipse data, process metrics can be used as predictors of defective code unit.

Hassan analyzed six open source projects to validate the hypothesis that the more complex the changes to a file, the higher the chances this file will contain faults [Has09]. He found two main results: (i) the number of prior faults is a better predictor for future faults than the number of prior modifications; (ii) the complexity of the code change is a better predictor for future faults than prior modifications or prior faults. D'ambros et al. also used process metrics as independent variable in their models [DLR12]. The authors proposed two new metrics called churn and entropy of source code metrics. Their results showed that churn and entropy of source code achieved the best adjusted $R^2$ and Spearman coefficient in four out of the five analyzed systems.

Typically, defect prediction models are used to identify defect-prone files or packages. Kamei et al. proposed a new approach for defect prediction called "Just-In-Time Quality Assurance" that focus on identifying defect-prone software changes instead of files or packages [KSA$^+$13]. Based on logistic regression, the models they propose identify whether or not a change is defect-prone using changes metrics, such as number of modified files, number of developers involved in the change, lines of code added and deleted, etc. They performed an empirical study with six open-source and five commercial systems to evaluate the performance of the models. The results showed an average precision of 34% and an average recall of 64%. However, they evaluate the models using a 10-fold cross-validation technique. On the other hand, cross-validation operates on a single time frame and therefore does not consider the temporal aspect.

Table 2.3: Empirical studies on process metrics and defect prediction

| Study | Indep. Variable | Model Technique | Results |
|---|---|---|---|
| [GKMS00] | Change Metrics | Linear Regression | Change metrics can be use as predictor of future faults. |
| [NB05b] | Change Metrics | Logistic Regression and Linear Regression | Code churn is able to predict system defect density. |
| [MPS08] | Change Metrics and Source Code Metrics | Logistic Regression and others | Change metrics can be used as defect predictors defective code unit. |
| [Has09] | Entropy of Changes | Linear Regression | The more complex the changes to a file, the higher the chance the file will contain faults. |
| [DLR12] | Churn and Entropy of Code Metrics | Linear Regression | Entropy and churn had a better performance than CK metrics. |
| [KSA+13] | Change Metrics | Linear Regression | Change metrics can to cover 62% of the defects. |

In this thesis, we trained our models using data from a time frame and validated them using data from future time frames.

### 2.3.3 Bug Finding Tools Approaches

Basically, these approaches use warnings reported by bug finding tools as early indicators of future defects, as summarized in Table 2.4. As we can observe, Nagappan and Ball described an experiment to measure the correlation between warnings reported by static analysis tools and defects [NB05a]. By using Spearman's test, the authors found a positive correlation between the density of warnings issued by the PREfix/PREfast tools and the density of pre-release defects detected in the Windows Server 2003. In addition, they rely on linear regression to build models to predict the ability of PREfast/PREfix to predict future defects. The results showed that bug finding tools can be used as early indicators of defects.

Zheng et al. analyzed three systems developed at Nortel Networks using three commercial static analysis tools: Gimpel's FlexeLint[2], Reasoning's Illuma[3], and Klockwork's inForce and GateKeeper[4] [ZWN+06]. They followed the GQM process to determine the economical implications of using static analysis tools. The authors showed that the number of warnings raised by static analysis tools can be a fairly good indicator of fault prone modules.

Wagner et al. evaluated the effectiveness of bug finding tools in two large systems [WAWS08]. In their work, they considered two tools: FindBugs and PMD. The

[2]http://www.gimpel.com/html/flex.htm.
[3]http://www.reasoning.com.
[4]http://www.klockwork.com.

Table 2.4: Empirical studies on warnings and defect prediction

| Study | Indep. Variable | Model Technique | Validation Technique |
|---|---|---|---|
| [NB05a] | PREfix/PREfast warnings | Linear regression | Spearman Correlation |
| [ZWN$^+$06] | Automated static analysis (ASA) | – | Spearman Correlation and Precision Measures |
| [WAWS08] | FindBugs and PMD warnings | – | Removed Warnings Rate and Spearman Correlation |
| [ASV11] | FindBugs and PMD warnings | – | Removed Warnings Rate |
| [CASV13] | FindBugs warnings | – | Precision and Recall |

goal was to assess the effectiveness of such tools to detect defects that occur in the field. For the first evaluated system, they did not find a single warning generated by FindBugs and PMD that could be related to a field defect. For the second system, they found a direct correspondence between four warnings and field defects. The authors concluded that bug finding tools are not effective to prevent field defects, since a small number of defects were detected.

Araujo et al. reported a study on the lifetime of the warnings reported by the FindBugs and PMD tools in five stable releases of the Eclipse platform [ASV11]. The authors classified a warning as relevant when it was removed some time after its first appearance in the system. They concluded that when the analysis is restricted to just warnings in the correctness category, 68.9% of the warnings were removed in later versions of the target system. Another conclusion was that PMD is not an effective tool to report relevant warnings, since only 26% of the warnings issued by this tool were classified as relevant.

#### 2.3.3.1 Static Correspondence between FindBugs Warnings and Defects

We conducted our own study to investigate whether the warnings issued by bug finding tools are related to defects [ACSV10, CASV13]. More specifically, the study evaluates whether the warnings issued by FindBugs are useful to predict the program elements that must be changed in order to remove field defects (bugs reported in bug tracking platforms). In this study, we analyzed three medium size systems: Rhino (a JavaScript interpreter with 31 KLOC that is developed as part of the Mozilla project), ajc (an AspectJ compiler, with around 63 KLOC), and Lucene (an information retrieval software library, with around 24 KLOC). We start by presenting the study setup and the tasks performed to collect data to assess the static correspondence between warnings and field defects. After that, we present the results and lessons learned.

**Study Setup:** We consider that there is a *static correspondence* between a warning $w$ reported by FindBugs and a field defect $d$ when $w$ is reported in the program elements that must be changed to fix $d$. For the first two systems, we relied on information available at the iBugs repository[5]. iBugs stores the source code before and after the correction of several defects reported by the users of the systems. The iBugs repository provides information about 32 issues (an issue may be a field defect, an improvement, or a new feature) reported by the Rhino's users. These issues were reported via Bugzilla[6], the bug tracking platform used by the Rhino's development team. In addition, the iBugs repository has information about 348 issues reported for the `ajc` compiler. For Lucene, we relied on the information available in the Jira bug tracking platform used by the Apache Foundation[7]. We considered information about 90 issues for Lucene.

We performed the following tasks to collect data to assess the static correspondence between warnings and field defects:

1. We filtered the issues that denote corrective maintenance tasks, since it makes no sense to expect a bug finding tool based on static analysis to predict the need of new features and improvements. For Rhino and `ajc`, we read and evaluated the text of each issue reported via Bugzilla. Our goal was to distinguish between issues that represent field defects and issues that in fact are improvements or new features. For Lucene, the filtering process was simpler, because Jira provides a search facility that allows to select only issues that are field defects. Table 2.5 reports the number of issues classified as corrective maintenance and as the other maintenance types. As can be observed, the percentage of corrective requests has been 50% (for Rhino), 66% (for `ajc`), and 33% (for Lucene).

Table 2.5: Classification of the maintenance requests

| **Maintenance types** | **Rhino** | | **ajc** | | **Lucene** | |
|---|---|---|---|---|---|---|
| | **Qty** | **%** | **Qty** | **%** | **Qty** | **%** |
| Corrective | 16 | 50 | 231 | 66 | 30 | 33 |
| Other types | 16 | 50 | 117 | 34 | 60 | 67 |
| Total | 32 | 100 | 348 | 100 | 90 | 100 |

2. We downloaded the source code before and after each issue we classified as corrective. For Rhino and `ajc`, the source code was retrieved directly from the iBugs repository. For Lucene, the source code was retrieved from the SVN version control platform, using the ID of the SVN transaction responsible for fixing a given

[5]http://www.st.cs.uni-saarland.de/ibugs
[6]http://www.bugzilla.org
[7]https://issues.apache.org/jira/secure/IssueNavigator.jspa

bug $b$ (this ID is provided by the Jira issue tracking platform). Moreover, we also retrieved from SVN the version of the system with an identifier equal to (ID-1), i.e., the version just before fixing the bug $b$.

3. We automatically compared the versions before and after fixing the considered field defects in order to find the defective methods. In our context, a defective method is a method changed to fix a field defect. We relied on a small parser for Java in order to calculate the changed methods[8]. By traversing the Abstract Syntax Tree (AST) generated by this parser, it was possible to retrieve the following information for each method: (a) signature, including name, parameters and return type; (b) a string representing the method's body. Using this information, we identified the methods changed from one version to another.

4. We executed FindBugs in its default configuration over the version before fixing the field defect. FindBugs generates a XML file with the total number of warnings of the system, the total number of warnings of each class and the location (field or method) of these warnings. We implemented a XML parser to read the warnings records and to collect the warnings located in the set of changed methods (as described in the previous item).

5. We evaluated the relevance of the warnings reported by FindBugs by measuring precision and recall of the warnings. By measuring precision, our goal was to provide information on the number of false positives raised by FindBugs, i.e. methods with warnings but that have not been changed to fix bugs. On the other hand, by measuring the recall our intention was to show information on the number of false negatives, i.e. the absence of warnings in methods changed to fix defects. First, we measured precision at the method level in the following way:

$$precision = \frac{number\ of\ changed\ methods\ with\ at\ least\ one\ warning}{number\ of\ methods\ with\ at\ least\ one\ warning}$$

To measure recall, we considered a method as relevant when it has been changed to fix a field defect. Moreover, we consider that FindBugs detects a relevant method when it raises at least one warning in such method. Based on these assumptions, we calculated recall in the following way:

$$recall = \frac{number\ of\ changed\ methods\ with\ at\ least\ one\ warning}{number\ of\ changed\ methods}$$

[8]http://code.google.com/p/javaparser/

**Results:** Table 2.6 shows the values measured for recall and precision for the three systems considered in this study. As we can observe, FindBugs reported a large number of warnings/KLOC for the three systems. On average, considering the versions analyzed in the study, FindBugs reported 3.6, 9.8, and 6.3 warnings/KLOC, for Rhino, `ajc`, and Lucene, respectively. In addition, FindBugs obtained a mean precision of 3.3%, 5.6%, and 6.8%, respectively. Therefore, the precision results were extremely low. This result indicates that FindBugs raises many warnings in methods that are not changed to fix bugs. Finally, FindBugs yielded an average recall of 3.8%, 6.9%, and 3.7%, for the systems Rhino, `ajc`, and Lucene, respectively. Such values indicate that the number of warnings reported by FindBugs is extremely low in the methods effectively changed to fix defects.

Table 2.6: Precision and recall

| | **Rhino (16 versions)** | | | |
|---|---|---|---|---|
| | **# Warnings** | **# Warnings/KLOC** | **Precision (%)** | **Recall (%)** |
| Max | 119 | 3.9 | 33.3 | 50.0 |
| Min | 101 | 2.6 | 0.0 | 0.0 |
| Mean | 112.6 | 3.6 | 3.3 | 3.8 |
| Median | 113.0 | 3.7 | 0.0 | 0.0 |
| Std Dev | 5.5 | 0.3 | 9.4 | 12.6 |

| | **ajc (206 versions)** | | | |
|---|---|---|---|---|
| | **# Warnings** | **# Warnings/KLOC** | **Precision (%)** | **Recall (%)** |
| Max | 938 | 11.7 | 100.0 | 100.0 |
| Min | 225 | 7.0 | 0.0 | 0.0 |
| Mean | 631.0 | 9.8 | 5.6 | 6.9 |
| Median | 813.0 | 9.9 | 0.0 | 0.0 |
| Std Dev | 267.9 | 0.7 | 19.6 | 22.4 |

| | **Lucene (30 versions)** | | | |
|---|---|---|---|---|
| | **# Warnings** | **# Warnings/KLOC** | **Precision (%)** | **Recall (%)** |
| Max | 205 | 6.9 | 100.0 | 100.0 |
| Min | 118 | 4.4 | 0.0 | 0.0 |
| Mean | 152.7 | 6.3 | 6.8 | 3.7 |
| Median | 152 | 6.3 | 0.0 | 0.0 |
| Std Dev | 30.8 | 0.5 | 20.5 | 18.2 |

**Lesson Learned**: Based on these results, we concluded that there is no *static correspondence* between field defects and warnings generated by FindBugs. In other words, the warnings generated by the tool would not have helped the maintainers to predict, understand, and remove the field defects evaluated in the study. The main reason for this result is the fact that the warnings reported by FindBugs are based on violations of coding practices (e.g., a class that implements `hashCode` but does not implement an `equals` method) and errors detected by data and control flow analysis (e.g., null pointer dereference). On the other hand, several field defects are related to logic errors (i.e., errors due to incorrect results). For example, in the specific case of Rhino and `ajc` most errors are due to source code in Javascript or AspectJ that is not processed as expected.

### 2.3.4 Application of Causality Tests in Software Maintenance

Canfora et al. were one of the first to investigate the use of causality tests in software maintenance [CCPC10]. They proposed the use of the Granger Causality Test to detect change couplings, i.e., software artifacts that are frequently modified together [Gra81]. They claimed that conventional techniques to determine change couplings fail when the changes are not "immediate" but due to subsequential commits. Therefore, they proposed to use the Granger Causality Test to detect whether past changes in an artifact $a$ can help to predict future changes in an artifact $b$. More specifically, they proposed the use of a hybrid change coupling recommender, obtained by combining Granger and association rules (the conventional technique to detect change coupling). After an study involving four open-source systems, they concluded that their hybrid recommender provides a higher recall than the two techniques alone and a precision in-between the two.

### 2.3.5 Critical Appraisal on Defect Prediction Approaches

For years, several works were conducted with the purpose of investigating relationships between software metrics and defects [BBM96, DLR10, NB05a, CASV13, GKMS00, Has09]. This fact demonstrates that, despite being an issue is of great importance to the software development process, defect prediction still remains an open problem. In other words, a defect prediction approach that can be adopted by real-world software development projects is still to be reached.

Moreover, most studies reported in the literature aiming to validate software metrics for predicting defects do not consider the idea of causality between software metrics and defects. Stated otherwise, they do not investigate whether the discovered relationships indicate cause-effect relations or whether they are mere statistical coincidences. As presented in Tables 2.2 and 2.3, the most common modeling techniques used by defect predictors are linear and logistic regressions. However, linear and logistic regressions do not imply causality [Ful94]. This means that when we identify a linear or logistic relationship between two variables, we can not conclude that one of the variables is the cause of (or directly affects) the other. More specifically, it is well known that regression models cannot filter out spurious correlations [Ful94].

Another issue we want to address concerns the output of the current defect prediction models. Currently, this output typically indicates the number or the existence of defects in a component in the future. The availability of this information is important to ensure software quality. However, predicting defects as soon as they are introduced in the source code, e.g., identifying the changes to a class that have more chances to generate defects, can be more useful for developers than simply identifying future occurrences of defects.

Finally, despite the interest and the increasing number of bug finding tools, there is still no consensus on the effective power of these tools to detect defects. Basically, studies aiming to assess the relationship between warnings reported by bug finding tools and bugs also do not consider the idea of causality. Typically, these studies are based on correlation tests such as the Spearman or Pearson correlation test [ACSV10, CASV13]. However, correlation does not imply causality. This means that when we identify a correlation between two variables, we can not conclude that one of the variables is the cause of the other. In other words, spurious correlations may also exist.

To contribute to tackle these issues, we propose in this thesis a defect prediction approach centered on more robust evidences towards causality between source code metrics (as predictors) and the occurrence of defects. The proposed approach differs from the presented studies with respect to three central aspects:

- To the best of our knowledge, the existing defect prediction approaches do not consider the idea of causality between software metrics and defects. Differently, our approach relies on causality tests to infer cause-effect relationships between source code metrics and defects.

- Typically, most studies evaluate their models in a single time frame. In contrast, we evaluated our approach in several life stages of the considered systems.

- Unlike common approaches for defect prediction, the models we propose do not aim to predict the number of defects of a class in a future time frame. Instead, our models trigger alarms that indicate changes to a class that have more chances to generate defects.

## 2.4 Final Remarks

In this chapter, we provided a discussion about software quality and defect prediction and present the software metrics commonly used to predict defects. We discussed the importance of the defect prediction area for software engineering, highlighting the relevance of identifying in advance those components of a software system that are more likely to fail. Particularly, we stated that defect prediction can be useful to determine where the software quality assurance resources must be allocated. Next, we presented the most common software metrics used to predict defects in object-oriented systems.

Finally, we presented the state-of-the-art in defect prediction and performed a critical appraisal of related work. In sections 2.3.1, 2.3.2, and 2.3.3, we discussed several works that investigate the relationships between software metrics and defects. In the following, we described a study that evaluates whether the warnings issued by a bug finding tool contribute to predict the program elements responsible for defects. The main criticism raised by us on theses studies concerns on how reliable is the relationship between software metrics and defects. The studies performed to evaluate defect prediction techniques generally do not investigate whether the discovered relationships indicate cause-effect relations or whether they are mere statistical coincidences.

# Chapter 3

# Prediction Techniques

In this chapter, we start first by describing the most common modeling techniques used by defect prediction approaches (Section 3.1). In Section 3.2, we present an example of a behavior that can not be captured by standard regressions. Next, we describe a precondition that Granger Causality Test requires the time series to follow and we present and discuss the test (Section 3.3). Finally, we describe and discuss another causality technique (Section 3.4).

## 3.1 Linear and Logistic Regression

The goal of linear regression is to describe the linear relationship between a dependent variable $y$ and one or more independent variables $(x_1, x_2, \cdots, x_k)$ [Tri06]. For models with a single variable $x$, the typical regression equation is expressed in the form $y = b_0 + b_1x$ and for models involving more variables $(x_1, x_2, \cdots, x_k)$, the general form of a multiple regression equation is $y = b_0 + b_1x_1 + b_2x_2 + \cdots + b_kx_k$.

Basically, regression equations can be used for predicting the value of the dependent variable, given some particular value of one or more independent variables. For example, in the context of defect prediction, the dependent variable is the number of defects in the source code of a given class and the independent variables are the values provided by software metrics for this class. Therefore, the ultimate goal is to predict the number of future defects in this class.

Unlike linear regressions, logistic regressions express the relationship between a binary dependent variable (i.e., the dependent variable can take the value 1 with a probability of success $\pi$, or the value 0 with probability of failure $1 - \pi$) and one or more independent variables $(x_1, x_2, \cdots, x_k)$ [HL00]. The general form of a multiple logistic regression is based on the following equation:

$$\pi(x_1, x_2, \cdots, x_k) = \frac{e^{b_0+b_1x_1+b_2x_2+\cdots+b_kx_k}}{1+e^{b_0+b_1x_1+b_2x_2+\cdots+b_kx_k}} \tag{3.1}$$

where $\pi$ is the probability of the presence of a particular property, the $x_is$ are the independent variables, and $b_is$ are the regression coefficients of the independent variables that are estimated through the maximization of a likelihood function. In the context of defect prediction, the independent variables are the values provided by software metrics and the binary dependent variable is the presence or absence of defects in a given class. Therefore, the goal is to discover the probability of this class having a fault.

The most common statistics used to evaluate the quality of linear regressions is called adjusted $R^2$ [Tri06]. An $R^2$ coefficient measures how well the multiple regression fits the sample data. On the other hand, the quality of multiple logistic regression models is usually evaluated by *deviance* ($D$) [HL00], which has the same role that the residual sum of squares has in linear regression. Basically, a $D$ coefficient denotes a measure of the lack of fit to the data in a logistic regression model.

## 3.2 Critical Appraisal on Regression

As stated in Section 2.3.1, the most common modeling technique used by defect prediction approaches is linear and logistic regressions. However, linear and logistic regressions do not imply causality [Ful94]. This observation means that when we identify a linear or logistic relationship between two variables, we can not conclude that one of the variables is the cause of (or directly affects) the other variable. In order to illustrate this issue, we simulated data about software metrics and defects for five classes (Class1, Class2, Class3, Class4, and Class5) of a hypothetical system with 100 versions. Figure 3.1 presents two graphs containing the simulated data. In the graph in the top we plotted the time series with the simulated values of the hypothetical software metric for each class and in the graph in the bottom we plotted the time series with simulated values of the number of defects for each class. As can be observed, an increase in the value of the metric impacted in the number of defects in all the five classes a few time units later. For example, an increase in the value of the metric for the class Class1 from 36 to 136 in the time unit 28 caused an increase in the number of defects from 4 to 9 in the time unit 36.

On the other hand, most defect prediction approaches based on regressions do not directly rely on historical values, but on current metrics and defects values. In other words, the regressions are performed based on values calculated for a single version. For example, defect prediction approaches that use regression would rely on

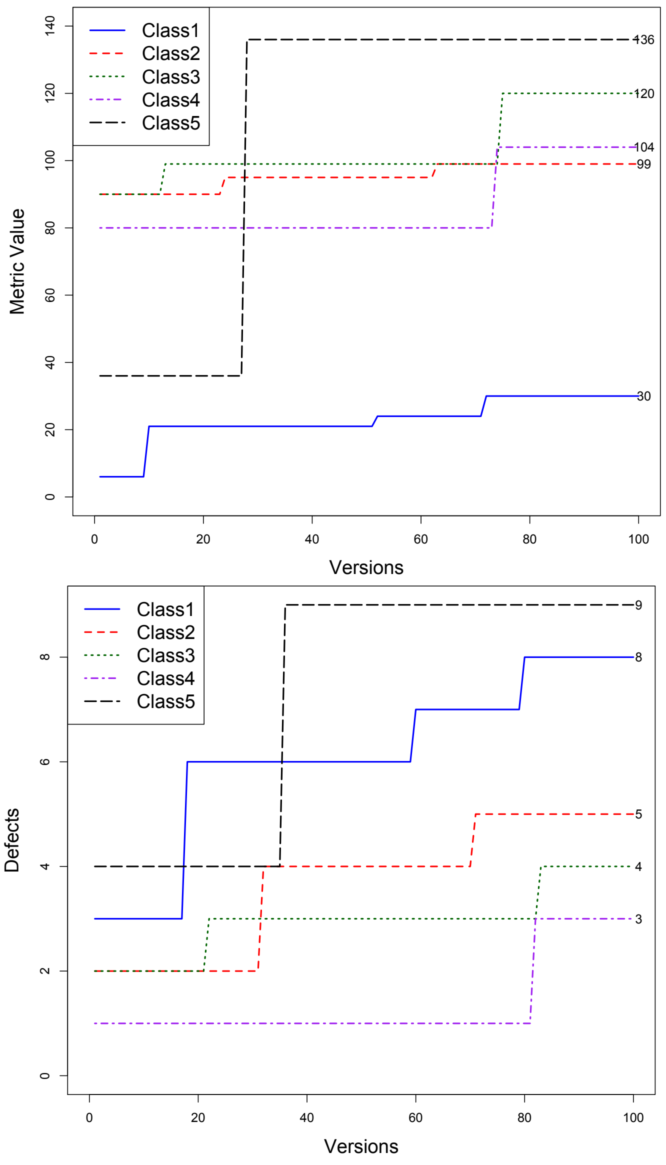


Figure 3.1: Time series of metric values and defects in a hypothetical system

values computed for the version 100 (the last version considered in the example). In the specific case of the data of Figure 3.1, the regression would be performed using the values (136, 120, 104, 99, 30) for the metrics and (9, 4, 3, 5, 8) for the defects. The regression for these values is not significant, i.e., there is no linear relationship between these values. Therefore, although there is a historical relationship between the metric values and the number of defects, i.e., historically, when the value of the metric increased the number of defects also increased, the regression was unable to capture such a relationship. On the other hand, specifically in these classes, the Granger Causality Test reported the existence of a cause-effect relationship between the time series of the metric and the time series of defects in all the five considered classes.

## 3.3 Granger Causality

In this section, we first describe a precondition that Granger requires the time series to follow (Section 3.3.1). Next, we discuss the test (Section 3.3.2).

### 3.3.1 Stationary Time Series

The usual pre-condition when applying forecasting techniques—including the Granger Test described in the next subsection—is to require a stationary behavior from the time series [Ful94]. In stationary time series, properties such as mean and variance are constant over time. Stated otherwise, a stationary behavior does not mean the values are constant, but that they fluctuate around a constant long run mean and variance. However, most time series of software source code metrics and defects when expressed in their original units of measurements are not stationary. The reason is intuitively explained by Lehman's Law of software evolution, which states that software measures of complexity and size tend to grow continuously [Leh80]. This behavior is also common in the original domain of Granger application, because time series of prices, inflation, gross domestic product, etc also tend to grow along time [Gra81].

An inherent feature in non-stationary series is the presence of unit root. A series has a unit root if the root of the characteristic equation of the series is 1. For example, the first order auto-regressive model for a time series $y_t = \alpha y_{t-1} + \epsilon_t$ has an unit root when $\alpha = 1$. Figure 3.2 shows the data of a simulated time series generated from the model $y_t = y_{t-1} + \epsilon_t$, $\epsilon_t \sim N(0, \sigma^2)$ and $y_0 = 2$, where $N$ implies normally distributed. As can be observed, we can not estimate the mean and variance, because the series has major variants.

Visual inspection of a time series rarely allows to distinguish between stationary and non-stationary behaviors. In addition, it is a tedious task to visually inspect series by series, when working with a large number of series. Thus, there are tests that verify the existence of unit root. Fuller proposed the Augmented Dickey-Fuller test to verify whether a time series has an unit root [Ful94]. The test consists of estimating an auto-regressive model of order $p$:

$$\Delta y_t = \mu + \alpha y_{t-1} + \sum_{i=1}^{p} \lambda_i \Delta y_{t-i} + \epsilon_t$$

and to use a $T$-test on the null hypothesis $H_0 : \alpha = 0$. A selection criterion for $p$ was

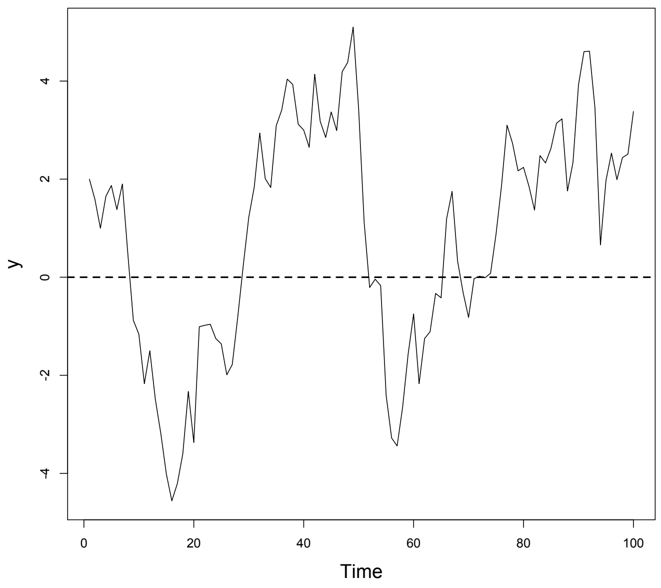


Figure 3.2: Simulated time series $y_t = y_{t-1} + \epsilon_t$ and $y_0 = 2$

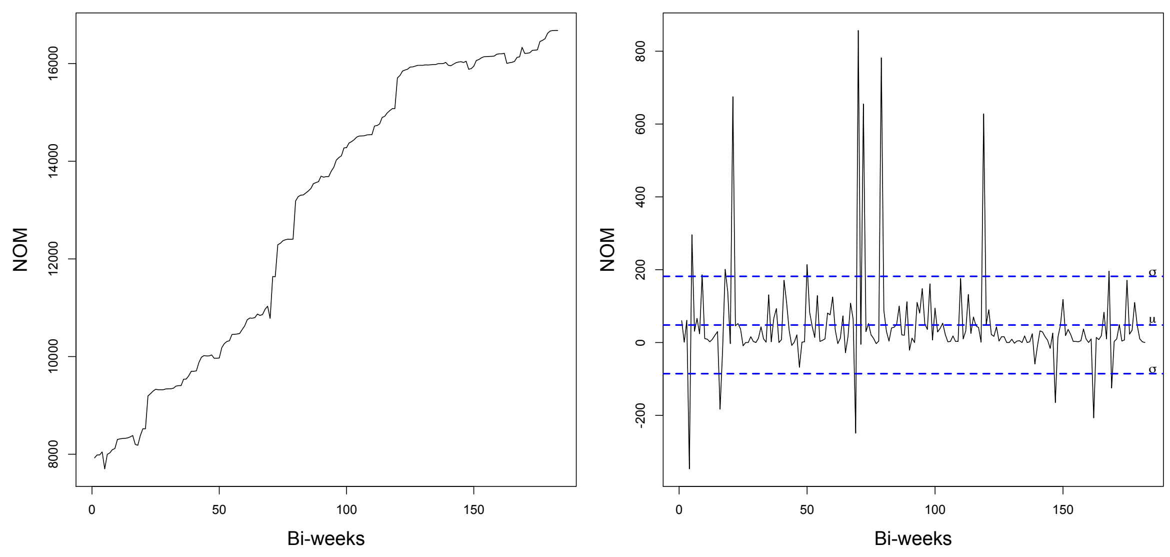
proposed by [Sch89], using the following formula:

$$p = int(12(T/100)^{1/4})$$

where $int(x)$ is the integer part of $x$. The econometric package *tseries* from the R statistical system implements the Augmented Dickey-Fuller test by means of the function *adf.test()*.

When the time series are not stationary, a common workaround is to consider not the absolute values of the series, but their differences from one period to the next. More specifically, suppose a time series $x(t)$. Its *first difference* $x'(t)$ is defined as $x'(t) = x(t) - x(t-1)$.

**Example #1:** To illustrate the notion of stationary behavior, we will consider a time series that represents the number of methods (NOM), extracted for the Eclipse JDT Core system, in intervals of bi-weeks, from 2001 to 2008. Figure 3.3(a) illustrates this series. As we can observe, the series is not stationary, since it has a clear growth trend, with some disruptions along the way. Figure 3.3(b) shows the first difference of NOM. Note that most values are delimited by a constant mean and variance. Therefore, NOM in first difference has a stationary behavior.



(a) Original NOM series (non-stationary behavior) (b) NOM in first difference (stationary behavior)

Figure 3.3: NOM for Eclipse JDT core

### 3.3.2 Granger Test

Testing causality between two stationary time series $x$ and $y$, according to Granger, involves using a statistical test—usually the F-Test—to check whether $x$ helps to predict $y$ at some stage in the future [Gra69]. If this happens, we can conclude that $x$ Granger-causes $y$. The most common implementation of the Granger Causality Test uses bivariate and univariate auto-regressive models. A bivariate auto-regressive model includes past values from the independent variable $x$ and from the dependent variable $y$. On the other hand, a univariate auto-regressive model considers only past values of the variable $y$.

To apply Granger, we must first calculate the following bivariate auto-regressive model [CCPC10]:

$$y_t = \alpha_0 + \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + \cdots + \alpha_p y_{t-p} + \beta_1 x_{t-1} + \beta_2 x_{t-2} + \cdots + \beta_p x_{t-p} + u_t \tag{3.2}$$

where $p$ is the auto-regressive lag length (an input parameter of the test) and $u_t$ is the residual. Essentially, $p$ defines the number of past values—from both $x$ and $y$—considered by the regressive models. Furthermore, Equation 3.2 defines a bivariate model because it uses values of $x$ and $y$, limited by the lag $p$.

To test whether $x$ Granger-causes $y$, the following null hypothesis must be rejected:

$$H_0 : \beta_1 = \beta_2 = \cdots = \beta_p = 0$$

This hypothesis assumes that past values of $x$ do not add predictive power to the regression. In other words, by testing whether the $\beta$ coefficients are equal to zero, the goal is to discard the possibility that the values of $x$ contribute to the prediction.

To reject the null hypothesis, we must first estimate the following auto-regressive univariate model (i.e., an equation similar to Equation 3.2 but excluding the values of $x$):

$$y_t = \gamma_0 + \gamma_1 y_{t-1} + \gamma_2 y_{t-2} + \cdots + \gamma_p y_{t-p} + e_t \tag{3.3}$$

Finally, to evaluate the precision of both models, we must calculate their residual sum of squares (RSS):

$$RSS_1 = \sum_{t=1}^{T} \hat{u}_t^2 \qquad RSS_0 = \sum_{t=1}^{T} \hat{e}_t^2$$

If the following test

$$S_1 = \frac{(RSS_0 - RSS_1)/p}{RSS_1/(T-2p-1)} \sim F_{p,T-2p-1}$$

exceeds the critical value of $F$ with a significance level of 5% for the distribution $F(p, T-2p-1)$, the bivariate auto-regressive model is better (in terms of residuals) than the univariate model. Therefore, the null hypothesis is rejected. In this case, we can conclude that $x$ causes $y$, according to the Granger Causality Test.

**Example #2:** In our previous Eclipse JDT Core example, we applied Granger to evaluate whether the number of public methods (NOPM), in the Granger sense, causes NOM. Although the common intuition suggests this relation truly denotes causality, it is not captured by Granger's test. Particularly, assuming $p = 1$ (the lag parameter), the F-Test returns a *p-value* of 0.32, which is superior to the defined threshold of 5%. To explain this lack of Granger-causality, we have to consider that variations in the number of public methods cause an immediate impact on the total number of methods (public, private, etc.). Therefore, Granger's application is recommended in scenarios where variations in the independent variable are reflected in the dependent variable after a delay (or lag).

**Example #3:** To explain the sense of causality captured by Granger in a simple and comprehensive way, suppose a new time series defined as:

$$NOM'(t) = \begin{cases} NOM(t) & \text{if } t \leq 5 \\ NOM(t-5) & \text{if } t > 5 \end{cases}$$

Basically, NOM' reflects with a lag of five bi-weeks the values of NOM. We reapplied Granger to evaluate whether NOPM causes NOM', in the Granger sense. In this case, the result was positive, assuming $p = 5$. Therefore, knowing the NOPM values at a given bi-week helps to predict the value of NOM'. Figure 3.4 illustrates the behavior of both series. For example, we can observe that just before bi-week 21 a significant increase occurred in the number of public methods. By knowing this information, one could predict an important increase in NOM' in the following bi-weeks. In fact, the figure shows that this increase in NOPM propagates to NOM' in few bi-weeks (we circled both events in the presented series).

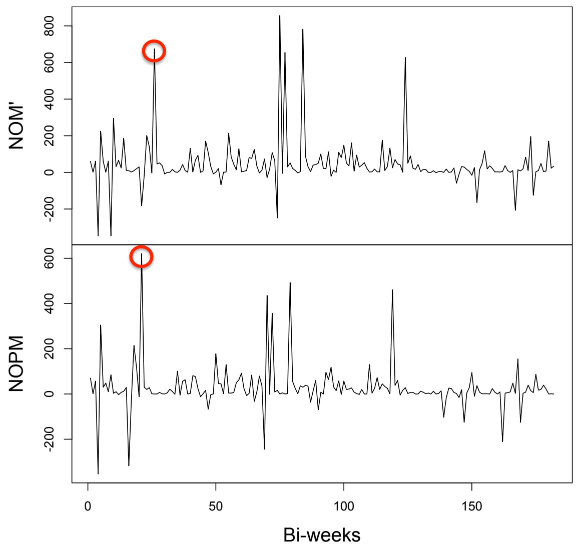


Figure 3.4: NOPM and NOM' time series. The increase in NOPM values just before bi-week 21 has been propagated to NOM' few weeks later

**Example #4:** To illustrate the application of Granger in a real example, Figure 3.5 shows the time series of LOC (lines of code) and defects for four classes of the Eclipse JDT Core system. These time series were created in intervals of bi-weeks from 2001 to 2008. In the figure, we circled the events in the time series of LOC that probably anticipated similar events in the time series of defects. For example, in the `SearchableEnvironmentRequestor` class (first series), the increase in LOC just before

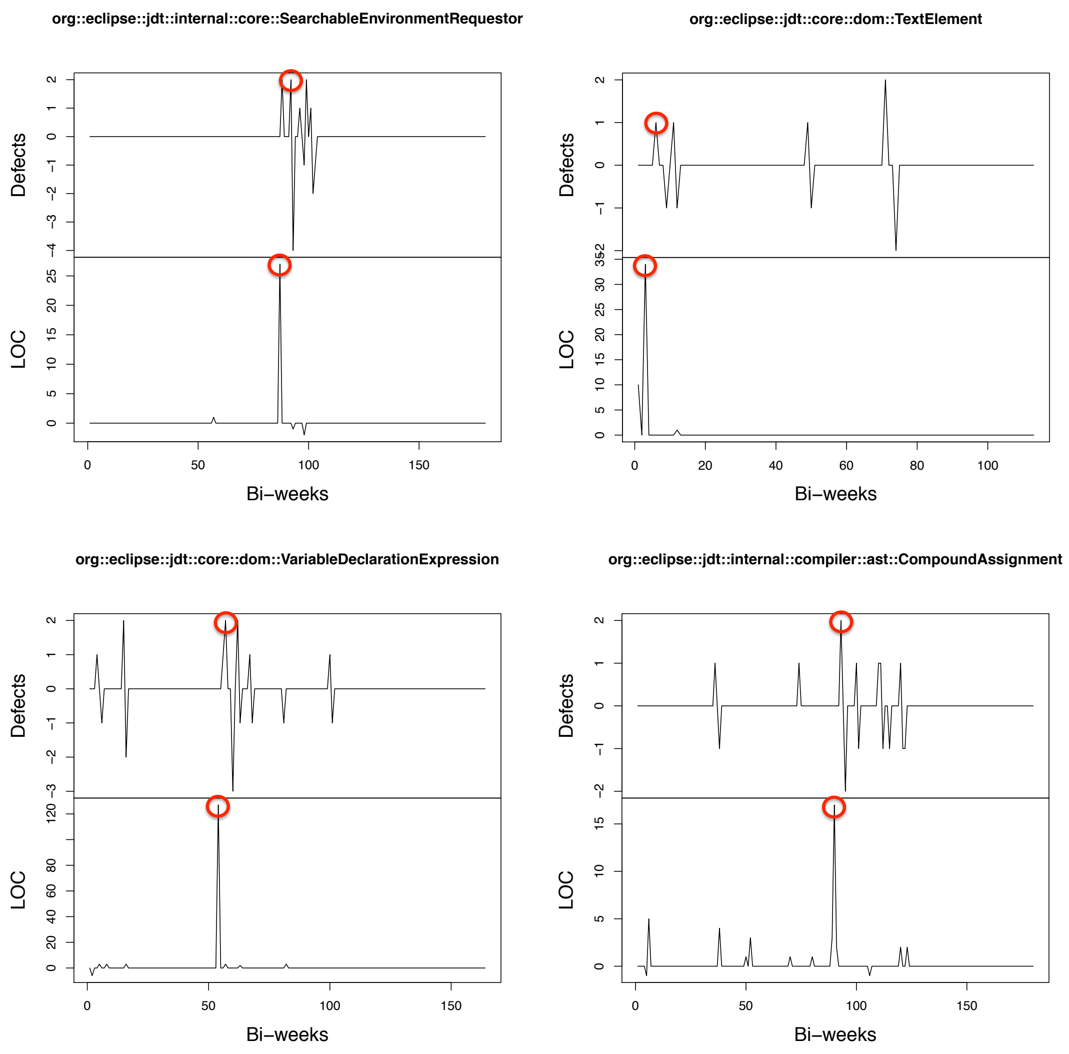


Figure 3.5: Examples of Granger Causality between LOC and defects.

bi-week 87 generated an increase in the number of defects few weeks later. In this classs specifically, a Granger-causality has been detected between LOC and defects, assuming $p = 3$.

## 3.4 Other Causality Techniques

Pearl proposes a probabilistic graphical model called bayesian network [Pea85]. Basically, bayesian network is a directed acyclic graph (DAG) whose vertices represent random variables and the edges represent the conditional dependencies among these variables. A distinguishing feature of this modeling technique is the possibility to model uncertainty and subjectivity dependencies. It allows to integrate subjective evaluations defined by experts and objective evaluations learned from data. More specifically, the graph structure that models the dependencies among variables is defined by experts and the probability distributions for each vertex is learned from data. Particularly, each vertex is associated with a prior probability (in case the vertex has no parents) or conditional probability (in case the vertex has parents). The conditional probabilities define the state of a variable given the combination of states of its parents.

Figure 3.6 illustrates a simple bayesian network (extracted from [Pea00]). It describes the dependency relationships among the variables $X_1$, $X_2$, $X_3$, $X_4$, and $X_5$. $X_1$ represents the seasons, $X_2$ whether it is raining, $X_3$ whether the sprinkler is on, $X_4$ whether the pavement is wet, and finally, $X_5$ whether the pavement is slippery. According to Pearl, a causal network is a bayesian network with the added property that the parents of each vertex are their direct causes. Therefore, in this case, the season has a direct effect on the rain or on the use of sprinklers. The rain and the sprinkler have a direct effect on the pavement to be wet. Finally, wet pavements are also slippery.

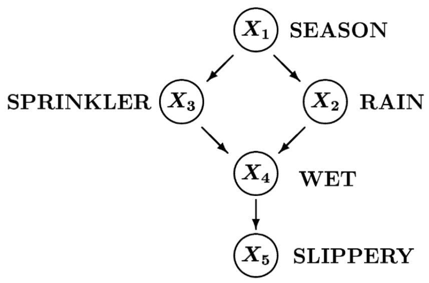


Figure 3.6: Bayesian network representing dependencies among five variables [Pea00]

Posteriorly, Pearl proposes a theory of inferred causation from the directed acyclic graph [Pea00, Ch. 2]. In this theory, a causal structure of a set of variables $V$ is a DAG in which each vertex is an element of $V$ and each edge represents a direct causal relationship among the corresponding variables. Basically, a variable $X$ is said to have a causal influence on a variable $Y$ if a directed path from $X$ to $Y$ exists in every *minimal causal structure* consistent with the data. *Minimal causal structures* are causal structures with less vertices and edges that are equally consistent with the data. Pearl proposes an algorithm to generate *minimal causal structures* and to induct causations, which is based on conditional independencies among the variables.

By comparing Granger Causality and Pearl Causality, we can identify two main differences: (i) Granger Causality is based on autoregressive models constructed from historical time series data while Pearl Causality is based on causal bayesian networks;

(ii) Granger Causality does not support a subjectivity dependencies while Pearl Causality allows to define dependencies by experts. Therefore, a future work could be the use of defect prediction models based on causal bayesian networks that allow the definition of dependencies by experts and to compare them with defect prediction models based on Granger Causality.

## 3.5 Final Remarks

In this chapter, we presented the modeling techniques commonly used by defect prediction approaches: linear and logistic regression. After that, we presented an example of a behavior that can not be captured by standard regressions. Next, we presented the Granger Causality Test, which is used by the proposed model to detect causal relationships between time series of source code metrics and time series of defects. Finally, we described the Pearl Causality and pointed out some differences between it and the Granger Causality.

# Chapter 4

# Dataset

In this chapter, we start by describing the original dataset used in this thesis (Section 4.1) and move to describe our extension of this dataset including time series of source code metrics and defects (Section 4.2). We then present detailed information on the provided time series (Section 4.3). Finally, we present related datasets (Section 4.4).

## 4.1 Original Dataset

The dataset used in this thesis is based on a dataset made public by D'Ambros et al. to evaluate defect prediction techniques [DLR10, DLR12]. This dataset includes temporal series for seventeen source code metrics, including number of lines of code (LOC) and the CK (Chidamber and Kemerer) metrics suite [CK94]. The metrics were extracted in intervals of bi-weeks for four well-known Java-based systems: Eclipse JDT Core, Eclipse PDE UI, Equinox Framework, and Lucene. Table 4.1 provides detailed information on this original dataset. In this table, column **Period** informs the time interval in which the metrics were collected by D'Ambros et al. In total, the dataset has 4,298 classes, each of them with at least 90 bi-weekly versions (which is equivalent to around three and a half years).

Table 4.1: Original dataset

| System | Period | Classes | Versions |
|---|---|---|---|
| Eclipse JDT Core | 2005-01-01 – 2008-05-31 | 1,041 | 90 |
| Eclipse PDE UI | 2005-01-01 – 2008-09-06 | 1,924 | 97 |
| Equinox Framework | 2005-01-01 – 2008-06-14 | 444 | 91 |
| Lucene | 2005-01-01 – 2008-10-04 | 889 | 99 |
| Total | - | 4,298 | 377 |

## 4.2 Extended Dataset

To conduct the research proposed in this thesis, we extended this dataset as follows: (a) by extracting again all source code versions and recalculating the source code metrics; (b) by almost doubling the number of source code versions included in this dataset, and (c) by creating time series of defects. We recalculated again the source code metrics because to calcutate them, we used different tools than the authors of the original dataset. Table 4.2 provides detailed information on our extended dataset. As can be observed, our extension has approximately twice the number of versions (723 versions) and 45% more classes (6,223 classes). Basically, we extended the original dataset to consider—whenever possible—the whole evolution history of the considered systems, starting from the first version available in their version repositories.

Table 4.2: Extended dataset

| System | Period | Classes | Versions |
|---|---|---|---|
| Eclipse JDT Core | 2001-07-01 – 2008-06-14 | 1,370 | 183 |
| Eclipse PDE UI | 2001-05-24 – 2008-04-03 | 3,478 | 180 |
| Equinox Framework | 2003-11-25 – 2010-10-05 | 615 | 180 |
| Lucene | 2002-06-22 – 2009-05-02 | 760 | 180 |
| Total | - | 6,223 | 723 |

Similarly to the original dataset, our extension does not include test classes. Test classes were discarded because they are not related to the core functionality of their systems and therefore they may statistically invalidate attempts to perform predictions. More specifically, we removed the directories and subdirectories whose name starts with the words “Test" or “test". The number of removed classes is as follows (for the last versions included in our dataset): 3,452, 208, 816, and 360 classes for Eclipse JDT Core, Eclipse PDE UI, Equinox Framework, and Lucene, respectively.

Furthermore, we consider a reduced number of source code metrics, as indicated in Table 4.3. More specifically, we reduced the number of source code metrics from seventeen to seven, for the following reasons:

- The seven metrics we selected already cover different properties of code, such as complexity (WMC), coupling (FAN-IN and FAN-OUT), cohesion (LCOM), and size (NOA, LOC, and NOM).
- The metrics related to inheritance—such as Depth of Inheritance Tree (DIT) and Number of Children (NOC)—usually do no present positive results regarding the Granger Causality Test, at least according to our previous study [CSV$^+$12].

It is important to highlight that eventual collinear relations between the considered source code metrics values do not have a major impact in our model. Basically, collinear pairs of metrics (like NOM and LOC, possibly) just tend to produce multiple alarms for the same defects, assuming that a Granger-causality is detected between them and defects. For this reason, we did not check for collinearity in our dataset.

Table 4.3: Metrics considered in our dataset

| | **Metrics** | **Description** | **Category** |
|---|---|---|---|
| 1 | WMC | Weighted methods per class | Complexity |
| 2 | LCOM | Lack of cohesion in methods | Cohesion |
| 3 | FAN-IN | Number of classes that reference a given class | Coupling |
| 4 | FAN-OUT | Number of classes referenced by a given class | Coupling |
| 5 | NOA | Number of attributes | Size |
| 6 | LOC | Number of lines of code | Size |
| 7 | NOM | Number of methods | Size |

## 4.2.1 Time Series of Source Code Metrics

We created time series of source code metrics for each class of the systems included in the dataset. To create such series, we extracted the source code versions of the system from its version control platform (SVN and Git) in intervals of bi-weeks during the time frame indicated in Table 4.2. We then used the Moose platform[1] to calculate the metrics values for each class of each considered version, excluding test classes. Particularly, we relied on VerveineJ[2]—a Moose application—to parse the source code of each version and to generate MSE files. MSE is the default file format supported by Moose to persist source code models. We extended the Moose platform with a routine to calculate LCOM, since the current version of Moose does not support this metric.

In a second step, we stored the computed metrics values in CSV files. Basically, for each system $S$ and each metric $M$, we created a CSV file whose lines represent the classes of $S$ and the columns represent the bi-weeks considered when extracting the versions of $S$. A cell $(c, t)$ in this file contains the value of the metric $M$, measured for the class $c$, in the bi-week $t$.

[1]http://www.moosetechnology.org

[2]http://www.moosetechnology.org/tools/verveinej

### 4.2.2 Time Series of Defects

In our terminology, we consider that bugs are failures in the observable behavior of systems. Bugs are caused by one or more errors in the source code, called defects [Sta90]. We count defects at the class level since our ultimate goal is to trigger alarms due to changes in classes. More specifically, each class changed to fix a given bug is counted as a defective class. Therefore, whenever mentioned that a system has $n$ defects in a given time frame, we are actually stating that we counted $n$ defective classes in this time frame (i.e., classes that were later changed to fix the defect). Classes with multiple defects related to the same bug are counted only once; on the other hand, defects in the same class but due to different bugs are counted separately. Finally, we do not consider open or non-fixed bugs.

The original dataset only provides information on the total number of defects reported for each class. Thus, in order to apply Granger we distributed this value along the bi-weeks considered in our evaluation. To create the time series of defects, the bugs—or more precisely, the maintenance requests—reported in the bug tracking platforms must be collected during the same time frame used to extract the source code versions. In a second step, each bug $b$ is linked to the classes changed to fix $b$, using the following procedure (which is also adopted in other studies on defect prediction [DLR10, ZPZ07, SZZ05b]):

1. Suppose that *Bugs* is the set containing the IDs of all bugs reported during the time frame considered in the analysis.

2. Suppose that *Commits* is the set with the IDs of all commits in the version control platform. Suppose also that $Cmts[c]$ and $Chg[c]$ are, respectively, the maintainer's comments and the classes changed by each commit $c \in Commits$.

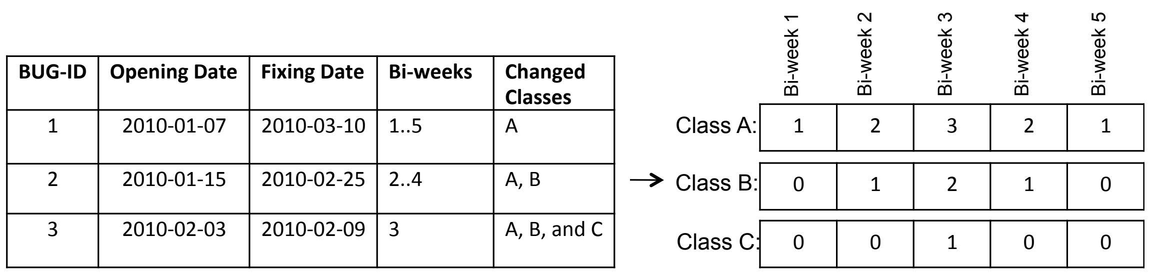
3. The classes changed to fix a given bug $b \in Bugs$ are defined as:

$$\bigcup_{\forall c \in \, Commits} \{ \; Chg[c] \;\mid\; substr(b, Cmts[c]) \; \}$$

   This set is the union of the classes changed by each commit $c$ whose textual comments provided by the maintainer includes a reference to the bug with ID $b$. The predicate $substr(s_1, s_2)$ tests whether $s_1$ is a substring of $s_2$.

Finally, suppose that in order to fix a given bug $b$ changes were applied to the class $C$. In this case, a defect associated to $b$ must be counted for $C$ during the period in which $b$ remained open, i.e., between the opening and fixing dates of $b$. More specifically, a defect is counted for the class $C$ at a time interval $t$ whenever the following conditions hold: (a) $b$ has been opened before the ending date of the time interval $t$; (b) $b$ has been fixed after the starting date of the time interval $t$.

Figure 4.1 shows an example regarding the extraction of a time series of defects with three bugs and three classes and spanning a time interval of five bi-weeks. The left table shows data on the bugs and the right figure shows the time series of defects. As we can observe, bug #1 was opened in 2010-01-07 (bi-week 1) and fixed in 2010-03-10 (bi-week 5). In order to fix this bug, changes were applied to the class A. In this case, a defect associated to bug #1 is counted for the class A during five bi-weeks.



| BUG-ID | Opening Date | Fixing Date | Bi-weeks | Changed Classes |
|---|---|---|---|---|
| 1 | 2010-01-07 | 2010-03-10 | 1..5 | A |
| 2 | 2010-01-15 | 2010-02-25 | 2..4 | A, B |
| 3 | 2010-02-03 | 2010-02-09 | 3 | A, B, and C |

$\rightarrow$

|  | Bi-week 1 | Bi-week 2 | Bi-week 3 | Bi-week 4 | Bi-week 5 |
|---|---|---|---|---|---|
| Class A: | 1 | 2 | 3 | 2 | 1 |
| Class B: | 0 | 1 | 2 | 1 | 0 |
| Class C: | 0 | 0 | 1 | 0 | 0 |

Figure 4.1: Example of extracting the time series of defects

To create the time series of defects, we followed the previously described methodology. We initially collected the issues (bugs) reported in the Jira and Bugzilla platforms (the bug tracking platforms of the considered systems) that meet the following conditions:

- Issues reported during the time interval considered by our dataset (as described in Table 4.2).
- Issues denoting real corrective maintenance tasks. Our goal was to distinguish between issues demanding corrective maintenance tasks and issues that in fact are requests for adaptive, evolutive or perfective maintenance. Jira has a field that classifies the issues as *bug*, *improvement*, and *new feature*. Therefore, we collected only issues classified as *bug*. On the other hand, Bugzilla is used mainly for corrective maintenance tasks (at least in the case of Eclipse Foundation systems). Despite that, some issues were classified as *enhancement* in the *Severity* field. Therefore, we also discarded them.
- Issues having *fixed* status. In other words, we discarded *open*, *duplicate*, *invalid*, and *incomplete* issues.

In a second step, we mapped the bugs to defects in classes and created the time series of defects for each class. Table 4.4 shows the number of bugs opened via Bugzilla or Jira for each of the systems. As can be observed, we collected a total of 6,614 bugs. This table also shows the number of defects that caused such bugs (i.e., number of classes changed to fix such bugs, according to the definition of defects, provided in Section 4.2.2), the number of defective classes (i.e., number of classes associated to at least one bug), and the average number of defects per bug. As can be observed, on average each bug required changes in 2.18 classes. Therefore, at least in our dataset, changes to fix bugs do not present a scattered behavior.

Table 4.4: Number of bugs, defects, and defects per bugs

| System | Bugs | Defects | Defective Classes | Defects/Bugs |
|---|---|---|---|---|
| Eclipse JDT Core | 3,697 | 11,234 | 833 | 3.04 |
| Eclipse PDE UI | 1,798 | 3,566 | 1,019 | 1.99 |
| Equinox Framework | 784 | 1,478 | 292 | 1.88 |
| Lucene | 335 | 615 | 157 | 1.83 |
| Total | 6,614 | 16,893 | 2,297 | 2.18 |

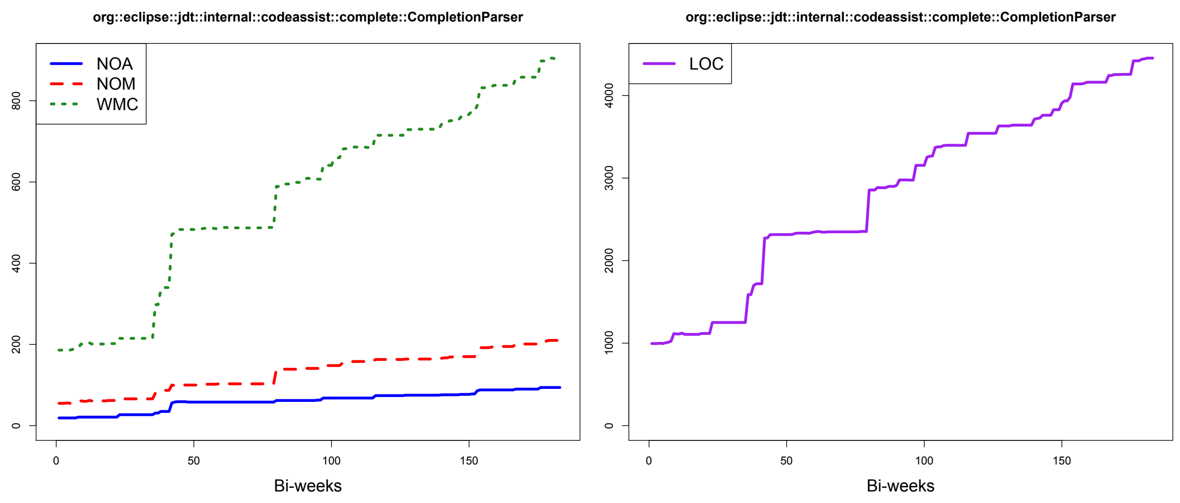
## 4.3 Provided Time Series

This section presents detailed information on the time series included in the dataset. For example, Table 4.5 describes some size properties we extracted from the time series of source code metrics. As we can observe, on total, we extracted 43,561 time series (seven time series of source code metrics for each class) from 6,223 classes (number of classes that has lived for at least one bi-week). In the first version (1st bi-week) of the considered systems, we counted 1,382 classes, totalizing 249 KLOC, 8,094 attributes, and 18,551 methods. Approximately seven years later (last bi-week), we counted 2,801 classes, around 613 KLOC, 20,942 attributes, and 42,619 methods.

Table 4.5: Size properties

| System | Classes | Series | Classes | | KLOC | | NOA | | NOM | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | 1st | Last | 1st | Last | 1st | Last | 1st | Last |
| JDT Core | 1,370 | 9,590 | 564 | 1,017 | 126 | 312 | 2,550 | 7,488 | 7,925 | 16,679 |
| PDE UI | 3,478 | 24,346 | 341 | 1,839 | 26 | 188 | 1,817 | 8,965 | 3,096 | 17,182 |
| Equinox | 615 | 4,305 | 156 | 313 | 60 | 56 | 2,624 | 2,338 | 4,703 | 4,201 |
| Lucene | 760 | 5,320 | 321 | 542 | 37 | 57 | 1,103 | 2,151 | 2,827 | 4,557 |
| Total | 6,223 | 43,561 | 1,382 | 2,801 | 249 | 613 | 8,094 | 20,942 | 18,551 | 42,619 |

Figure 4.2 shows examples of source code time series provided in the dataset. More specifically, Figure 4.2(a) shows the time series with the values of three metrics (NOA, NOM, and WMC) collected for the `CompletionParser` class of the `codeassist` package in the Eclipse JDT Core system. Figure 4.2(b) describes the evolution of this same class in terms of number of lines of code. As we can observe in these figures, this class had a considerable growth in terms of size (NOA, NOM, and LOC) and complexity (WMC) during its life.



(a) NOA, NOM, and WMC time series (b) LOC time series

Figure 4.2: Time series for a Eclipse JDT class

Figure 4.3 shows the time series of defects and the accumulated defects for the same class. As we can observe, this class has had more than one hundred defects during its life. Moreover, we counted a significative number of defects in the bi-week 102—around 15 defects. Therefore, the extracted time series show that this class is constantly involved in corrective maintenance tasks since the beginning of its life. Moreover, for this class specifically, there is a strong correlation between size and complexity metrics and defects. In other words, the higher is the size and the complexity of this class, the higher is the number of corrective maintenance tasks.

Figure 4.4(a) shows the evolution of the number of lines of code of the systems in our dataset. As we can observe, the JDT Core, PDE UI and Equinox systems had a linear growth trend. On the other hand, the Lucene system had a sharp drop in the number of lines of code in the bi-week 66. This event is associated with the removal of the source code of the LARM sub-project from the GIT repository on 2004-12-18, as expressed in the following commit message: *"LARM has left the building... and has*

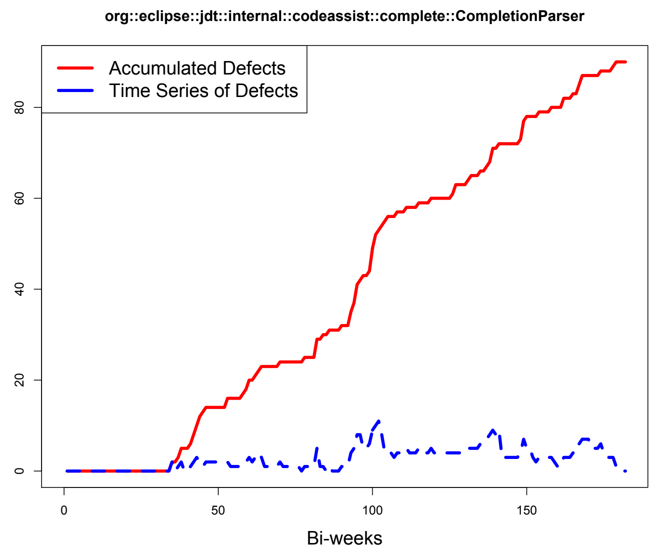


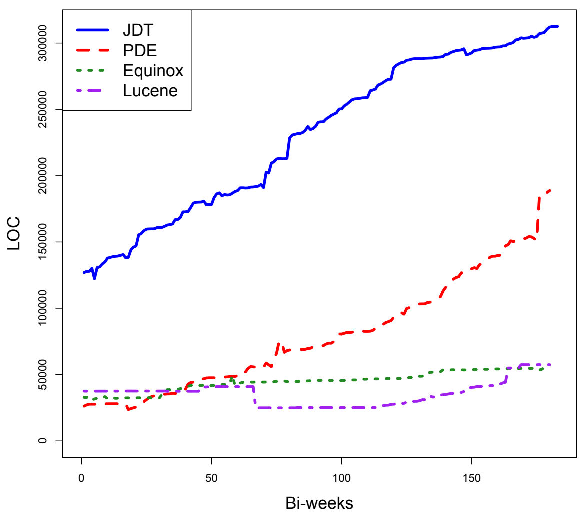
Figure 4.3: Defects time series (for a Eclipse JDT class)

*been living at larm.sf.net for a long time now"*. After this event, Lucene's source code continued increasing with a linear trend. Based on these figures, we can observe that the Lehman's Law of "Continuing Growth" is valid for the systems in our dataset, at least in terms of size.

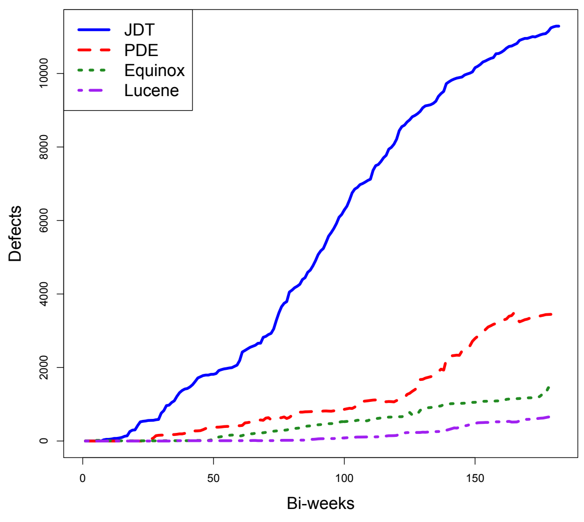
Finally, Figure 4.4(b) shows the behavior of the systems in terms of accumulated defects over the period of analysis. As we can observe, in the initial bi-weeks the classes of the systems had no defects. As time passed, the number of defects increases in all systems without a stabilizing trend. The growth of the Eclipse JDT Core system stands out from the rest. Its last version (bi-week 183) has 1,017 classes with 11,278 accumulated defects.

## 4.4 Related Datasets

Helix is a dataset that provides temporal information on the values of source code metrics [VLJ10]. However, Helix lacks information on the number of defects at the level of classes (time series of defects) and some of the metrics included in our dataset, such as cohesion and coupling metrics. The Qualitas Corpus is another well-known dataset for empirical studies in software engineering [TAD$^+$10]. It contains information on 111 systems, but it only provides information on the evolution of 14 systems (only one with more than 70 versions). The Qualitas.*class* Corpus is a compiled version of the Qualitas Corpus [TMVB13]. It provides compiled Eclipse projects for all systems included in the original Qualitas Corpus. On the other hand, in our dataset all systems have at



(a) LOC time series



(b) Defects time series

Figure 4.4: Time series for the systems in our dataset

least 180 versions. Moreover, both datasets do not include temporal information on the values of source code metrics.

## 4.5 Final Remarks

In this chapter, we described our dataset including time series of source code metrics and defects. Our goal with this dataset is to assist an initial study on the causal relationships between source code metrics and defects (Chapter 5) and to construct and validate our defect prediction model (Chapter 6). Our dataset includes three systems from the Eclipse project and one system from the Apache Foundation with a total of 6,223 classes and 49,784 time series (including time series of source code metrics and defects). Therefore, we claim that this dataset includes a credible number of classes, representing real-world and non-trivial applications, with a consolidated number of users and a relevant history of defects.

The dataset with the time series used in this thesis is publicly available at:

http://aserg.labsoft.dcc.ufmg.br/jss2013

This dataset and the know-how we achieved on the extraction of data from version control platforms motivated us to create another dataset called COMETS [CMGV13, RCM+12]. COMETS (Code Metrics Time Series) is a dataset of source code metrics collected from several systems to support empirical studies on source code evolution. The dataset includes information on the evolution of 10 Java-based systems, including Eclipse JDT Core, Eclipse PDE UI, Equinox, and Lucene. The COMETS dataset is available for download at:

$\boxed{\text{http://aserg.labsoft.dcc.ufmg.br/comets}}$

# Chapter 5

# Feasibility Study

In this chapter, we describe a first study designed to investigate the feasibility of applying Granger to detect causal relationships between time series of source code metrics and defects. Particularly, our goal is to evaluate whether there are causal relationships between source code metrics values and the number of defects in the classes of object-oriented systems. For this purpose, we propose an algorithm including preconditions to check whether source code metrics time series are in fact useful to forecast changes in time series of defects, at least according to Granger.

We start by describing a real example of application of the Granger Causality Test in the context of defect prediction (Section 5.1). Next, we present the algorithm including preconditions required to apply the Granger Test (Section 5.2). Section 5.3 reports and discuss the results and lessons learned after this study. Finally, we present the final remarks (Section 5.4).

## 5.1 Granger Test Application

In this section, we describe a real example of the application of Granger in the context of defect prediction. For this purpose, we use two time series (LOC and defects) extracted from the class `jdt.core.dom.SuperConstructorInvocation` of the Eclipse JDT system. Figure 5.1 illustrates the behavior of both time series. In this figure, the $x$-axis represents 164 bi-weeks (the time frame used to create the time series) and the $y$-axis represents the values of LOC and defects. As we can observe, the figure shows that an increase in the variable LOC impacts in the variable defects few weeks later. Specifically for this class, the Granger Test reported the existence of causality between LOC and defects with a lag size of 5, since the $p$-value found was equal to 0, and therefore lower than the significance level defined for the study ($\alpha = 0.05$).

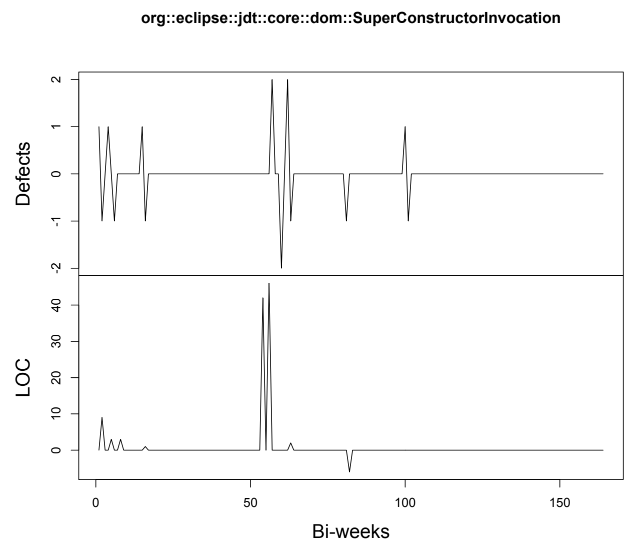


Figure 5.1: Series LOC and number of defects for a class of the Eclipse JDT system

As described in Section 3.3.2, the Granger Test evaluates whether the bivariate autoregressive model is better than the univariate model with a certain significance level. Particularly, for the example in Figure 5.1, the test evaluates whether the bivariate autoregressive model including temporal values of LOC and defects is better than the univariate autoregressive model including only values of defects. This evaluation is performed by analyzing the model's residuals. These residuals are the difference $(y-\hat{y})$ between an observed sample value $y$ and the value $\hat{y}$ predicted by the regression equations. Table 5.1 presents the residual sum of squares (RSS) and Adjusted $R^2$ for the univariate and bivariate autoregressive models constructed from the data in Figure 5.1.

Table 5.1: RSS and Adjusted $R^2$ for the bivariate and univariate autoregressive models

| **Model** | **RSS** | **Adjusted $R^2$** |
|---|---|---|
| Bivariate | 7.77 | 0.56 |
| Univariate | 14.58 | 0.20 |

As can be observed in the Table 5.1, the RSS of the bivariate model is smaller than in the univariate model. In addition, the adjusted coefficient of determination $R^2$ of the bivariate model is greater than the univariate model. However, to confirm that the bivariate model is better than the univariate model, the Granger Test relies on a $F$-test. For this particular case, this test returned a $p$-value less than 0.05. Therefore, we can conclude that LOC causes defects, at least according to Granger Test.

## 5.2 Applying the Granger Test

To apply the Granger Causality Test to identify causal relations on all time series of source code metrics and defects in our dataset, we propose the following algorithm:

**Algorithm 1** Granger Test

```
 1: for all c ∈ Classes do
 2:     d = Defects[c];
 3:     if d_check(d) then
 4:         for n = 1 → NumberOfMetrics do
 5:             m = M[n][c];
 6:             if m_check(m) then
 7:                 granger(m, d);
 8:             end if
 9:         end for
10:     end if
11: end for
```

In this algorithm, `Classes` is the set of all classes of the system (line 1) and `Defects[c]` is the time series with the number of defects (line 2). The algorithm relies on function `d_check` (line 3) to check whether the defects in the time series `d` conform to the following preconditions:

- P1: The time series must have at least $k$ values, where $k$ represents the minimum size that a series must have to be considered by the prediction model. Therefore, time series that only existed for a small proportion of the time frame considered in the analysis—usually called dayfly classes [Lan01, RCM$^+$12]—are discarded. The motivation for this precondition is the fact that such classes do not present a considerable history of defects to qualify their use in predictions.

- P2: The values in the time series of defects must not be all null (equal to zero). Basically, the goal is to discard classes that never presented a defect in their lifetime (for instance, because they implement a simple and stable requirement). The motivation for this precondition is that it is straightforward to predict defects for such classes; probably, they will remain with zero defects in the future.

- P3: The time series of defects must be stationary, which is a precondition required by Granger, as reported in Section 3.3.1.

Suppose that a given class `c` has passed the previous preconditions. For this class, suppose also that `M[n][c]` (line 5) is the time series with the values of the `n`-th source code metric considered in the study, $1 \leq$ `n` $\leq$ `NumberOfMetrics`. The algorithm relies on function `m_check` (line 6) to test whether the time series `m`—a time series with metrics values—conforms to the following preconditions:

- P4: The time series of source code metrics must not be constant. In other words, metrics time series whose values never change must be discarded, since variations in the independent variables are the key event to observe when computing Granger causality.

- P5: The time series of source code metrics must be stationary, as defined for the defects series.

Finally, for the time series `m` (source code metrics) and `d` (defects) that passed preconditions P1 to P5, function `granger(m,d)` checks whether `m` Granger-causes `d` (line 7). As described in Section 3.3.2, Granger is sensitive to the lag selection. For this reason, in the proposed algorithm the test is not applied for a single lag value, but several times, with the lags ranging from 1 to $l$. In this way, we consider that a metric `m` is a Granger-cause of defects in a given class `c` whenever one of the tested lags return a positive result.

## 5.3 Setup and Results

In this section, we describe the study designed to investigate the feasibility of using the Granger Test for discovering causal relations between source code metrics and defects. More specifically, we focus on the following questions:

1. How many time series pass the preconditions related to defects (preconditions P1, P2, P3)?

2. How many time series pass the preconditions related to source code metrics (preconditions P4 and P5)?

3. How many classes present positive results on the Granger Test?

4. What is the number of defects potentially covered by our approach?

5. What are the lags that most led to positive results regarding Granger-causality?

To answer these questions, we used the entire dataset described in Chapter 4. Therefore, we analyzed 6,223 classes, 16,893 defects, and approximately 50,000 time series of source code metrics and defects, with a maximum size of 183 bi-weeks (JDT

Core) and a minimal size of 180 bi-weeks (PDE UI, Equinox Framework, and Lucene).

**Parameters Setting:** An important decision when applying the Granger Test is setting the parameters used in the preconditions, as described in Section 5.2. In practice, we decided to set such parameters in the following way:

- *Minimum size:* We defined that the classes should have a lifetime of at least 30 bi-weeks (approximately one year). Our goal is to select classes with a sufficient history of defects that qualify their use in predictions (and therefore to tackle the cold-start problem that typically happens when making predictions based on historical data [SPUP02]).
- *Maximum lag:* We computed the tests using a lag ranging from one to six. To set this maximum lag, we analyzed the time interval between the opening and fixing dates of the bugs in our dataset. On average, 84% of the bugs were fixed within six bi-weeks.
- *Significance level:* We computed the tests using a significance level of 95% ($\alpha = 0.05$). We counted as causality the cases where the $p$-value obtained by applying the F-Test was less than or equal to $\alpha$, i.e., $p$-value $\leq 0.05$.

**Tool Support:** The algorithm described in Section 5.2 was implemented in the R statistical system. We considered all times series in first difference (see Section 3.3.1) to maximize the number of stationary time series—a precondition to apply the Granger Test. To identify stationary time series, we relied on function *adf.test()* of the *tseries* package. This function implements the Augmented Dickey-Fuller Test for stationary behavior [Ful94]. More specifically, this function receives as parameters the time series to be checked and a lag. Particularly, we relied on the default lag suggested by the function. To apply Granger, we used function *granger.test()* of the *msbvar* package.

## 5.3.1 Preconditions on Time Series of Defects

The algorithm proposed in Section 5.2 first checks whether the defects times series pass the preconditions P1, P2, and P3 using function `d_check`. Table 5.2 shows the percentage and the absolute number of classes that survived these preconditions. We can observe that 57% of the classes survived precondition P1 (lifetime greater than 30 bi-weeks) and that 34% of the classes survived both P1 and P2 (at least one defect in their lifetime). Finally, our sample was reduced to 31% of the classes after applying the last precondition (test for stationary behavior). In summary, after checking the preconditions P1, P2, and P3, our sample was reduced significantly.

Table 5.2: Percentage and absolute number of classes conforming to preconditions P1, P2, and P3

| System | P1(%) | Classes | P1+P2(%) | Classes | P1+P2+P3(%) | Classes |
|---|---|---|---|---|---|---|
| Eclipse JDT Core | 80 | 1090 | 59 | 811 | 57 | 779 |
| Eclipse PDE UI | 45 | 1582 | 26 | 918 | 23 | 788 |
| Equinox Framework | 65 | 397 | 44 | 271 | 36 | 219 |
| Lucene | 59 | 450 | 19 | 142 | 17 | 131 |
| Total | 57 | 3,519 | 34 | 2,142 | 31 | 1,917 |

### 5.3.2 Preconditions on Time Series of Source Code Metrics

The second step of the algorithm described in Section 5.2 relies on function `m_check` to evaluate the preconditions P4 and P5. Considering only the classes passing preconditions P1, P2, and P3, Table 5.3 shows the percentage of source code time series that passed preconditions P4 and P5. As defined in Section 5.2, precondition P4 states that the time series must not be constant and P5 requires the series to be stationary. By observing the values in Table 5.3, we conclude that constant time series are common for some metrics. For example, for LCOM, FAN-IN, and NOA approximately 40% of the considered classes presented a constant behavior (column Total). Furthermore, we can observe that the number of series with non-stationary behavior—even when considering the first differences—is not negligible. For example, for WMC, 84% of the series survived P4, but only 74% survived P5. In summary, after checking the preconditions P4 and P5, our sample of time series of source code metrics was reduced to 65%.

### 5.3.3 Defects Covered by Granger

After checking the proposed preconditions, the algorithm computes function `granger` to check the existence of Granger-causality. Table 5.4 shows for each class `c` the number of tests with a positive result considering the series `M[n][c]` and `Defects[c]`, where `M[n][c]` is one of the seven series of metrics for a given class `c` ($1 \leq n \leq 7$) and `Defects[c]` is the series of defects for this class. For example, for Eclipse JDT Core, 62% of the classes have no Granger-causality relationship between their defects series and one of the metrics series (Table 5.4, first line). Stated otherwise, in 38% of the classes in the Eclipse JDT Core (i.e., 521 classes), we were able to detect a Granger-causality relation between the series of defects and at least one of the seven series of

Table 5.3: Percentage of time series conforming successively to preconditions P4 and P5

| **Metric** | **JDT Core (%)** | | **PDE UI (%)** | | **Equinox (%)** | | **Lucene (%)** | | **Total (%)** | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **P4** | **P4+P5** | **P4** | **P4+P5** | **P4** | **P4+P5** | **P4** | **P4+P5** | **P4** | **P4+P5** |
| LCOM | 70 | 66 | 53 | 43 | 64 | 53 | 63 | 59 | 62 | 54 |
| WMC | 88 | 84 | 79 | 64 | 84 | 72 | 85 | 84 | 84 | 74 |
| FAN-IN | 60 | 57 | 47 | 39 | 50 | 43 | 76 | 73 | 55 | 49 |
| FAN-OUT | 76 | 72 | 80 | 69 | 76 | 67 | 75 | 73 | 78 | 70 |
| NOA | 61 | 57 | 60 | 51 | 62 | 53 | 59 | 55 | 61 | 54 |
| LOC | 94 | 91 | 92 | 73 | 90 | 79 | 95 | 89 | 93 | 82 |
| NOM | 80 | 75 | 76 | 62 | 76 | 66 | 79 | 76 | 77 | 69 |
| Total | 76 | 72 | 70 | 57 | 72 | 62 | 76 | 73 | 73 | 65 |

metrics; in around 9% of the classes Granger returned a positive result for a single series of metrics, and so on. In the remaining three systems—Eclipse PDE UI, Equinox, and Lucene—the percentage of classes where the test found a Granger-causality connection between metrics and defects was 12% (411 classes), 20% (124 classes), and 10% (78 classes), respectively. In summary, our sample was reduced considerably to 18% (1,134 classes) of its original size after applying Granger.

Table 5.4: Percentage and absolute values of classes with $n$ positive results for Granger

| **n** | **JDT Core** | | **PDE UI** | | **Equinox** | | **Lucene** | |
|---|---|---|---|---|---|---|---|---|
| | % | Classes | % | Classes | % | Classes | % | Classes |
| 0 | 62 | 849 | 88 | 3,067 | 80 | 491 | 90 | 682 |
| 1 | 9 | 122 | 4 | 141 | 5 | 31 | 2 | 16 |
| 2 | 7 | 99 | 2 | 71 | 3 | 18 | 2 | 13 |
| 3 | 6 | 78 | 2 | 55 | 4 | 23 | 2 | 13 |
| 4 | 6 | 77 | 2 | 53 | 4 | 22 | 1 | 9 |
| 5 | 4 | 53 | 1 | 45 | 3 | 20 | 2 | 13 |
| 6 | 4 | 49 | 1 | 36 | 1 | 7 | 1 | 11 |
| 7 | 2 | 43 | 0 | 10 | 0 | 3 | 0 | 3 |
| Total | 100 | 1,370 | 100 | 3,478 | 100 | 615 | 100 | 760 |

Finally, it is fundamental to check the number of defects in this subset of 1,134 classes. Table 5.5 shows the following results: number of classes, number of Granger positive classes (column GPC), number of bugs we initially collected, number of defects that caused such bugs, and number of defects detected in our subset of 1,134 classes (column DGC). More specifically, considering the classes with at least one positive result for Granger, Table 5.5 shows that 73% of the defects collected in our dataset were detected in such classes. Therefore, by combining the results in Tables 5.4 and 5.5, we conclude that our preconditions and the Granger results reduced our sample to 18% of its original size. However, such classes concentrate 73% of the defects in our dataset. Considering that there are many bugs not related to variations in source code metrics, it is natural to expect that our coverage would be significantly less than 100%. On the other hand, an average coverage of 73% shows that it is at least feasible to rely on Granger to predict defects in software systems.

Table 5.5: Classes, Granger positive classes (GPC), number of bugs, number of defects, number of defects in Granger positive classes (DGC)

| System | Classes | GPC | Bugs | Defects | DGC | DGC/Defects |
|---|---|---|---|---|---|---|
| Eclipse JDT Core | 1,370 | 521 | 3,697 | 11,234 | 8,781 | 78% |
| Eclipse PDE UI | 3,478 | 411 | 1,798 | 3,566 | 2,391 | 67% |
| Equinox Framework | 615 | 124 | 784 | 1,478 | 766 | 52% |
| Lucene | 760 | 78 | 335 | 615 | 462 | 75% |
| Total | 6,223 | 1,134 | 6,614 | 16,893 | 12,400 | 73% |

As previously described, the Granger tests were calculated using a significance level of 95% ($\alpha = 0.05$). In other words, we counted as a Granger-causality the cases where the $p$-value obtained by applying the Granger Test was less than or equal to $\alpha$, i.e., $p$-value $\leq 0.05$. Table 5.6 shows the percentage of tests with a positive result distributed in intervals of 1%. As can be observed, approximately 70% of the tests with a positive result returned a $p$-value less than 0.01 for all considered systems.

### 5.3.4 Lags Considered by Granger

It is well known that the Granger Test is sensitive to the lag selection [Gra81]. For this reason, as described in Section 5.2, we do not fix a single lag, but calculate the test successively for each pair of series, with the lags ranging from one to six. Whenever one of such lags returns a positive result, we assume the existence of Granger-causality.

Table 5.7 shows the lags that were most successful in returning positive results. When multiple lags returned causality, we chose the one with the lowest $p$-value. As we can note, we achieved different results for each system. For Eclipse JDT Core, 33% of the Granger-causalities were established for a lag equal to six bi-weeks. For Eclipse PDE UI and Equinox, the most successful lag was equal to one bi-week. For Lucene, the distribution was almost uniform among the six lags.

Table 5.6: Granger positive $p$-values

| | **JDT Core (%)** | **PDE UI (%)** | **Equinox (%)** | **Lucene (%)** |
|---|---|---|---|---|
| $0.04 < p\text{-value} \leq 0.05$ | 5 | 5 | 3 | 2 |
| $0.03 < p\text{-value} \leq 0.04$ | 5 | 5 | 3 | 2 |
| $0.02 < p\text{-value} \leq 0.03$ | 5 | 7 | 6 | 6 |
| $0.01 < p\text{-value} \leq 0.02$ | 10 | 8 | 8 | 6 |
| $0.00 < p\text{-value} \leq 0.01$ | 65 | 70 | 70 | 64 |
| Total | 100 | 100 | 100 | 100 |

Table 5.7: Percentage of lags with a positive result for Granger-causality (highest values in bold)

| **Lag** | **JDT Core** | **PDE UI** | **Equinox** | **Lucene** |
|---|---|---|---|---|
| 1 | 15 | **30** | **40** | 19 |
| 2 | 17 | 15 | 11 | 12 |
| 3 | 14 | 15 | 18 | 14 |
| 4 | 11 | 12 | 10 | 18 |
| 5 | 10 | 12 | 11 | 17 |
| 6 | **33** | 16 | 10 | **20** |
| Total | 100 | 100 | 100 | 100 |

We can interpret such results as follows. First, changes were made to the considered systems (which we will call event A). Such changes have an impact in the values of the metrics considered in our study (event B). Frequently, such changes also introduced defects in the source code (event C) and some of them became bugs reported in the system's bug tracking platform (event D). In this description, events A, B, and C can be considered as happening at the same time and they are succeeded by event D. Essentially, we rely on Granger to show the existence of causality between events C and D. According to this interpretation, Granger's lag is the typical distance between such events in the time. Therefore, the results in Table 5.7 suggest that in the case of the Eclipse JDT Core and Lucene most bugs were perceived by the developers in six bi-weeks. In contrast, for the Eclipse PDE UI and Equinox, this interval was of just one bi-week, in most of the cases.

To summarize, when applying the Granger Test to uncover causal relations between source code metrics and defects, it is important to run the tests with various lags. The reason is that the time between the inception of a defect in the source code and its perception by the maintainers as a bug can vary significantly.

**Lesson Learned:** Based on our findings in this initial study, we can conclude that our approach for defect prediction shows to be feasible in practice. Even reducing our sample to 18% of the classes after applying the preconditions and the Granger Test, it was possible to cover 73% of the defects in our dataset.

## 5.4 Final Remarks

In this chapter, we presented a study to evaluate the feasibility of our approach based on causality tests. The contributions of this study are: (a) an algorithm and a set of preconditions to mine for Granger-causality relationships between software quality metrics and defects at the class-level; and (b) a detailed discussion on the results and lessons learned after using this algorithm to mine for causalities between time series of source code metrics and time series of defects. After using Granger to mine for causalities between time series of metrics (extracted from source code versions) and time series of defects (extracted from the bugs history), we were able to associate to the historical values of metrics the causes of 73% of the defects reported for the systems considered in our dataset. Therefore, we claim that our approach based on causality tests seems to be feasible to be used in practice.

# Chapter 6

# Predicting Defects Using Granger

In this chapter, we propose an approach for predicting defects using causality tests. In Section 6.1, we describe the steps to construct the proposed model. We then describe a procedure to identify the thresholds in metrics variations that may lead to defects (Section 6.2). Next, we present an evaluation of the proposed model in Section 6.4. Finally, we discuss potential threats to the validity (Section 6.5).

## 6.1 Proposed Approach

The ultimate goal of our approach is to predict defects using a model centered on Granger-causality relations between source code metrics (independent variables) and defects (dependent variable). Our approach relies on historical data such as bug histories (extracted from bug tracking platforms) and source code versions (extracted from version control platforms). From this data, we create time series of source code metrics and defects for the classes of a target system. Next, we rely on the Granger Causality Test for inferring causal relations between the time series of metrics and defects. After that, we build a defect prediction model that triggers alarms when changes made to the target system have high chances of producing defects.

As illustrated in Figure 6.1, we propose the following steps to build a defect prediction model:

1. As described in Section 4.2.1, we create time series of source code metrics for each class of the target system. To create such series, source code versions of the target system are extracted from its version control platform in a predefined time interval (e.g., bi-weeks). After that, the values of the considered source code metrics are calculated for each class of each extracted version.

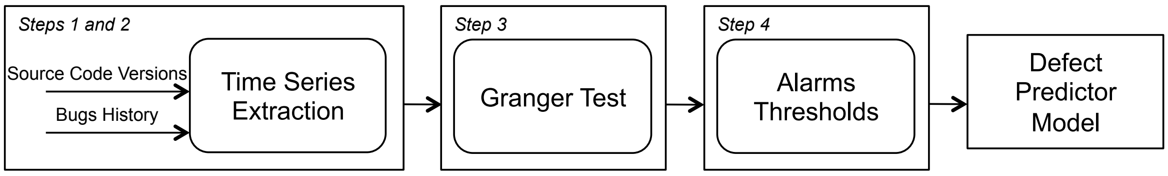


Figure 6.1: Steps to build a model for defect prediction

2. As described in Section 4.2.2, we create a time series with the number of defects in each class of the target system from the bugs history. Basically, we map the bugs reported in bug tracking platforms to their respective commits using the bug identifiers. Next, the files changed by such commits are used to identify the classes changed to fix the respective defects (i.e., the defective classes).

3. Following the methodology described in Section 3.3, we apply the Granger Causality Test considering the metrics and defects time series. More specifically, Granger is responsible for identifying causal relations on time series of source code metrics and defects.

4. As a distinguishing aspect of our approach, we identify thresholds for variations in metrics values that may contribute according to Granger to the occurrence of defects. More specifically, we build a model that relies on such thresholds to alert developers about future defects whenever a risky variation in the values of a metric happens due to changes in the system.

In the next section, we describe the proposed approach to identify alarms thresholds used by our model to trigger alarms.

## 6.2 Thresholds to Trigger Alarms

As described in Section 3.3.2, Granger Causality Test identifies whether an independent variable $x$ contributes to predict a dependent variable $y$ at some stage in the future. However, the test does not establish the thresholds for relevant variations of the values of $x$ that may impact $y$. Therefore, this step aims to calculate the thresholds used by our model to trigger alarms, as follows:

1. For each time series of source code metrics that Granger returned a positive result, we compute the positive variations in the series values, by subtracting the

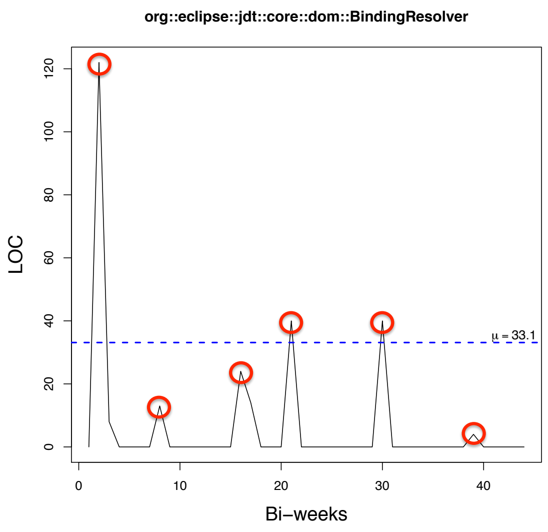


Figure 6.2: Example of a threshold for the LOC metric

values at consecutive bi-weeks[1].

2. A threshold to trigger alarms for a given class $C$ and metric $m$ is the arithmetic mean of the variations of $m$ computed for $C$, as defined in the previous step.

Figure 6.2 shows a time series for one of the classes of the Eclipse JDT Core system where Granger returned a positive result between the values of LOC and defects. In this figure, we circled the positive variations used to calculate the alarms thresholds. As can be observed, the threshold for the class `BindingResolver` is 33.1, which is the arithmetic mean of the values we circled. The proposed defect prediction model relies on this threshold to alert maintainers about future defects in this class. More specifically, an alarm is triggered for future changes adding at least 34 lines of code to this class.

## 6.3 Defect Prediction Model

Figure 6.3 illustrates the inputs and the output of the proposed prediction model. Basically, the model receives as input two values for a given source code metric $m$, $m_v$ and $m_{v'}$, where $m_v$ is the value of the metric regarding a class $C$ that was just changed

[1]We decided to compute the positive variations because they typically indicate a degradation in the internal quality of the source code, which may influence the occurrence of future defects. Therefore, at least in principle, it does not make sense to trigger alarms in cases where the variations in the metric values are negatives, i.e., when the source code quality improves.

to fix a bug. Moreover, $m_{v'}$ is the value of the metric in the previous version of $C$ in the version control platform. The proposed model verifies whether $m$ Granger-causes defects in $C$ and whether the difference $(m_v - m_{v'})$ is greater or equal to the threshold identified for variations in the metric values. When both conditions hold, the model triggers an alarm. Basically, such alarm indicates that, according to the Granger Test, similar variations in the values of this metric in the past resulted in defects.

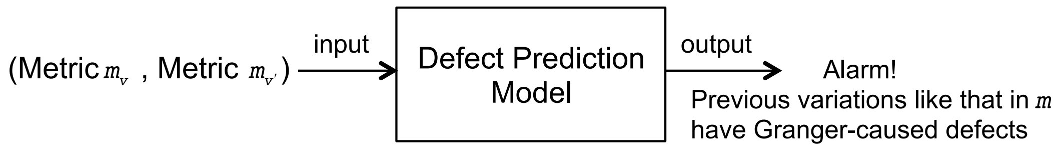

(Metric $m_v$ , Metric $m_{v'}$)
input
Defect Prediction Model
output
Alarm!
Previous variations like that in $m$ have Granger-caused defects

Figure 6.3: Defect prediction model

With this model in hand, a maintainer before making a commit in the version control platform with changes to a given class can verify whether such changes may lead to defects. If the model triggers an alarm for a given class warning about future occurrences of defects, the maintainer can for example perform extra software quality assurance activities in this class (e.g., unit testing or a detailed code inspection) before executing the commit.

## 6.4 Evaluation

In the feasibility study reported in Section 5, we concluded that, even by reducing our sample to 18% of the classes after applying the preconditions and the Granger Test, it was possible to cover 73% of the defects reported for the systems considered in our dataset. Motivated by such positive results, we decided to conduct a second study to evaluate our model for triggering defects alarms. The study described in this section also relies on the dataset described in Chapter 4. More specifically, this study aimed to answer the following research questions:

**RQ1: What is the precision and recall of our approach?** With this question, we want to investigate whether our defect prediction models provide reasonable levels of precision and recall.

**RQ2: How does our approach compares with the proposed baselines?** Our aim with this question is to analyze the precision of our approach when compared to three baselines. The first baseline does not consider the results of the Granger Test, the second one does not consider both the preconditions defined in Section 5.2 and the results of the Granger Test and the third one uses simple linear regression as prediction technique.

**RQ3: What is the impact in the precision results of using other functions (different from the mean) for triggering alarms?** As defined in Section 6.2, the thresholds used to trigger alarms for a given class $C$ and metric $m$ is the mean of the positive variations of the values of $m$ computed for $C$. Therefore, our goal with this question is to investigate whether alternative descriptive statistics functions—such as minimum, first quartile, median, third quartile, and maximum—provide better precision values than the mean when used to trigger alarms.

**RQ4: Regarding their severity, what types of bugs are typically predicted by the proposed models?** Our goal with this research question is to investigate whether our models tend to predict with higher accuracy some particular categories of bugs, in terms of severity. Particularly, we intend to rely on the severity categories informed by the users of the Bugzilla and Jira tracking platforms.

In this section, we start by presenting the methodology followed in our evaluation (Section 6.4.1). After that, we provide answers and insights for our research questions (Section 6.4.2). Finally, we discuss threats to validity (Section 6.5).

### 6.4.1 Evaluation Setup

We performed the following steps to answer the proposed research questions:

1. We divided the time series (considered in their first differences) in two parts. We used the first part (training series) to build a defect prediction model and the second part (validation series) to validate this model. Moreover, we defined that the time series start in the first bi-week with a reported defect. For example, for the Eclipse JDT Core, our training series start in the bi-week 8, because we have not found defects in the previous bi-weeks. We also defined the size of the validation series as including exactly 18 bi-weeks, i.e., approximately six months (which is a time frame commonly employed in studies on defect prediction [DLR10, HPH+09, SZZ06]). For example, for a time series with 50 bi-weeks in the Eclipse JDT Core, we discarded the first seven bi-weeks (since the first defect appeared only in the 8th bi-week). We used the next 25 bi-weeks for training, and the 18 remaining bi-weeks for validation.

2. We created a defect prediction model for each system according to the methodology described in Sections 5.2, 6.2, and 6.3. More specifically, we first checked the preconditions and applied the Granger Test considering the source code metrics (independent variables) and the defects (dependent variable) time series. Next, we identified the thresholds for variations in the metrics values that may have contributed to the occurrence of defects. Finally, we created a prediction model that triggers defects alarms.

3. We defined three baselines to evaluate the models constructed in the Step 2. In these baselines, the way to calculate the thresholds is exactly the one used by our approach, i.e., the arithmetic mean of the positive variations of the metrics. However, they differ on the preconditions and on the use of the Granger Test, as described next:

   a) The first baseline is a model created using time series meeting the preconditions P1 to P5, but that does not consider the results of the Granger Test. Therefore, variations in any source code metrics that respect the preconditions can trigger alarms (i.e., even when a Granger-causality is not detected). The purpose of this baseline is to check whether the Granger Test contributes to improve the precision of the proposed models.

   b) The second baseline considers time series meeting only precondition P1 (i.e., this model does not consider the preconditions P2 to P5 and the results of the Granger Test). We preserved precondition P1 because it is fundamental to remove classes with a short lifetime that do not help on reliable predictions. An alarm is triggered by variations in any metric that respects the first precondition, even when a Granger-causality is not detected. The central purpose of this second baseline is to evaluate the importance of the preconditions in the proposed model.

   c) The third baseline considers time series meeting the preconditions P1 to P5, but instead of applying the Granger Test, we created a simple linear regression model. More specifically, this model is composed by linear equations that correlate each source code metric separately (independent variable) and the defects time series (dependent variable). We checked the significance of the individual coefficients of the regressions in order to identify if a given metric is effective to predict the occurrence of defects. Therefore, alarms are only triggered due to variations in the metrics whose individual coefficients are statistically significant ($\alpha = 0.05$). The main goal of this third baseline is to evaluate whether the Granger Test is more effective than linear regressions to express relations between source code metrics and defects.

4. To evaluate the accuracy of our models, we used classification as validation technique. Table 6.1 illustrates the confusion matrix used to classify the alarms triggered by our models.

Table 6.1: Confusion matrix

| | **Alarm** | |
|---|---|---|
| **New Defect** | **Yes** | **No** |
| **Yes** | True Positive (TP) | False Negative (FN) |
| **No** | False Positive (FP) | True Negative (TN) |

Based on this confusion matrix, we calculated the precision and recall measures. Particularly, precision evaluates whether the alarms issued by the model are confirmed by defects. To calculate precision, we used only the validation time series, i.e., series with values not considered during the model construction phase. An alarm issued in a given bi-week $t$ is classified as a *true alarm* when a new defect is identified at most six bi-weeks after bi-week $t$. On the other hand, an alarm is classified as a *false alarm* when there is no new defect at most six bi-weeks after bi-week $t$. Therefore, we calculate precision in the following way:

$$precision = \frac{number\ of\ true\ alarms\ (TP)}{number\ of\ true\ alarms\ (TP) + number\ of\ false\ alarms\ (FP)}$$

Conversely, recall measures whether the alarms triggered by our approach cover the defects found in Granger positive classes. To calculate recall we checked whether the occurrences of defects in the validation series were preceded by an alarm. More specifically, we checked whether a defect in a given bi-week $t$ was preceded by an alarm in at most six bi-weeks before $t$. We calculate recall in the following way:

$$recall = \frac{number\ of\ true\ alarms\ (TP)}{number\ of\ true\ alarms\ (TP) + number\ of\ defects\ without\ alarms\ (FN)}$$

Figure 6.4 shows a hypothetical example of calculation of precision and recall for the alarms triggered by the proposed model. In this figure, balls represent classes. Classes with new defects (black balls) are to the left of the straight line while classes without new defects (white balls) are to the right. Suppose that the proposed model triggered 9 defect alarms for the 9 classes within the circle. As can be observed, from these 9 alarms, 6 were classified as true alarm (TP) and 3 were classified as false alarm (FP). Thus, the precision of the alarms triggered by the model is of 66% (6/9). On the other hand, from the 15 classes with new defects (TP + FN), the proposed model triggered alarms only for 6 classes (TP). Thus, the recall of the proposed model is of 40% (6/15).

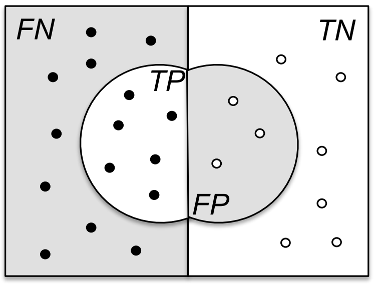


Figure 6.4: Hypothetical example of calculation precision and recall

5. We repeated Steps 1 to 4 for several time frames, i.e., for multiple training and validation time series. Our main goal is to evaluate the proposed approach in different life stages of the considered systems. Figure 6.5 illustrates the time frames considered for the Eclipse JDT Core. As presented, we created and validated 144 different models, considering different time frames. The first time frame has 30 bi-weeks, including 12 bi-weeks to build the model (training time series) and 18 bi-weeks to validate the model (validation time series). To generate a new model, we extended the previous training series in one bi-week. For example, the second time frame has 31 bi-weeks, the third one has 32 bi-weeks, etc. Finally, the last time frame has 174 bi-weeks. For the systems Eclipse PDE UI, Equinox Framework, and Lucene, we created and validated 125, 145, and 115 models, respectively.

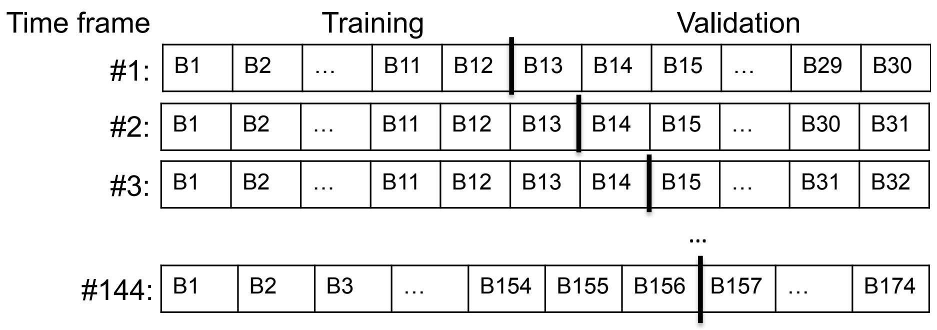

Time frame
Training
Validation
#1: B1 B2 ... B11 B12 | B13 B14 B15 ... B29 B30
#2: B1 B2 ... B11 B12 B13 | B14 B15 ... B30 B31
#3: B1 B2 ... B11 B12 B13 B14 | B15 ... B31 B32
...
#144: B1 B2 B3 ... B154 B155 B156 | B157 ... B174

Figure 6.5: Training and validation time series (Eclipse JDT Core)

### 6.4.2 Results

In this section, we provide answers to our research questions.

#### 6.4.2.1 RQ1: What is the precision and recall of our approach?

To address this research question, we followed Steps 1 to 5 described in Section 6.4.1. Therefore, we created and validated models for each time frame of the systems considered in this evaluation. Our main goal was to evaluate the proposed approach in different life stages of the considered systems. The tables in Figure 6.6 shows the values we measured for true alarms, precision, recall, and F-measure for the considered systems. Considering all time frames, the tables also report the following results: maximum value (Max), the top 5%, 10%, and 20% values, minimum value (Min), average, median, and standard deviation (Std Dev). As can be observed, our approach reached an average precision ranging from 28% (Eclipse PDE UI) to 58% (Eclipse JDT Core) and a median precision ranging from 31% (Eclipse PDE UI) to 58% (Eclipse JDT Core). Furthermore, some particular models presented high precisions, 90%, 60%, 100%, and 88%, for the Eclipse JDT Core, Eclipse PDE UI, Equinox and Lucene, respectively.

| Measure | TA | Pre | Rec | F |
|---|---|---|---|---|
| Max | 168 | 90% | 68% | 70% |
| Top 5% | 118 | 88% | 48% | 60% |
| Top 10% | 95 | 82% | 40% | 55% |
| Top 20% | 82 | 67% | 32% | 42% |
| Min | 2 | 27% | 7% | 12% |
| Mean | 56 | 58% | 24% | 33% |
| Median | 49 | 58% | 23% | 32% |
| Std Dev | 34 | 14% | 13% | 14% |

(a) Eclipse JDT Core (144 models)

| Measure | TA | Pre | Rec | F |
|---|---|---|---|---|
| Max | 36 | 60% | 44% | 38% |
| Top 5% | 33 | 41% | 36% | 33% |
| Top 10% | 31 | 40% | 33% | 31% |
| Top 20% | 27 | 36% | 29% | 30% |
| Min | 1 | 6% | 6% | 7% |
| Mean | 16 | 28% | 24% | 24% |
| Median | 16 | 31% | 23% | 26% |
| Std Dev | 11 | 11% | 7% | 7% |

(b) Eclipse PDE UI (125 models)

| Measure | TA | Pre | Rec | F |
|---|---|---|---|---|
| Max | 21 | 100% | 31% | 44% |
| Top 5% | 9 | 88% | 25% | 40% |
| Top 10% | 8 | 80% | 22% | 35% |
| Top 20% | 7 | 75% | 19% | 27% |
| Min | 1 | 22% | 5% | 8% |
| Mean | 5 | 53% | 13% | 20% |
| Median | 4 | 46% | 12% | 20% |
| Std Dev | 3 | 21% | 7% | 10% |

(c) Equinox Framework (145 models)

| Measure | TA | Pre | Rec | F |
|---|---|---|---|---|
| Max | 16 | 88% | 52% | 50% |
| Top 5% | 16 | 80% | 50% | 50% |
| Top 10% | 14 | 78% | 45% | 48% |
| Top 20% | 13 | 67% | 43% | 44% |
| Min | 0 | 0% | 0% | 10% |
| Mean | 8 | 51% | 31% | 36% |
| Median | 7 | 48% | 30% | 37% |
| Std Dev | 5 | 19% | 13% | 10% |

(d) Lucene (115 models)

Figure 6.6: True alarms (TA), precision (Pre), recall (Rec), and F-measure (F)

In general terms, we can conclude that our approach reached reasonable levels of precision in many life stages of the considered systems. This result is a distinguishing contribution of our evaluation, since defect prediction approaches typically analyze a single time frame [CASV13, KSA$^+$13, HPH$^+$09, DLR10, GDPG12]. For example, D'Ambros *et al.* created and validated their defect prediction models for the Eclipse JDT Core for a single time frame (2005-01-01 to 2008-06-17) [DLR10]. For the same system, we created and validated defect prediction models for 144 time frames achieving an average precision of 58%.

On the other hand, specifically for the Eclipse PDE UI system, our approach obtained an average precision of just 28%. Probably, this result was due to the low mapping rate between bugs and commits in this system. While for Eclipse JDT Core, Equinox, and Lucene we obtained a mapping rate of approximately 70%, Eclipse PDE UI reached a mapping rate around 46% (i.e., from the 3,913 bugs reported on the bug tracking platform, only 1,798 were linked to a commit on the version control platform).

We can also observe that some of the evaluated models triggered a significant number of true alarms. For example, for the system Eclipse JDT Core, the maximum number of true alarms triggered by a given model was 168 (for the model constructed in the time frame 49). Probably, this result is explained by a major maintenance activity in the system during the validation period of this model. We measured on average 277 classes changed per bi-week in this particular validation period, while this rate considering the entire period of analysis is 218. Figure 6.7 illustrates some validation time series where an alarm triggered by this model was later confirmed by the occurrence of defects. In this figure, we circled the true alarms issued by the model.

Despite such encouraging results regarding precision, our approach presented an average recall ranging from 13% (Equinox) to 31% (Lucene) and a median recall ranging from 12% (Equinox) to 30% (Lucene). In practice, this result shows that we were not able to cover all defects in all life stages of the considered systems. We argue that the main reason is the fact that there is a large spectrum of bugs that can be reported for any system. Probably, some types of bugs are less impacted by variations in the values of the source code metrics. For example, we can mention bugs related to usability concerns, internationalization, and JavaDoc documentation.[2] Despite this fact, we achieved reasonable levels of recall in particular time frames. For example, for the Eclipse JDT, Eclipse PDE UI, Equinox, and Lucene systems, the maximum values for

[2]As an example, we can mention the following bug reported for the Eclipse JDT Core system: *"Bug 10495 - typo in ASTNode::MALFORMED javadoc, 'detcted' should be 'detected'"*. Because JavaDocs are comments in the source code, a class was changed to fix this bug and a respective defect was included in our defects time series. However, it is not feasible to suppose that bugs like that can be predicted. In fact, an alarm was never raised by our models for this particular bug.

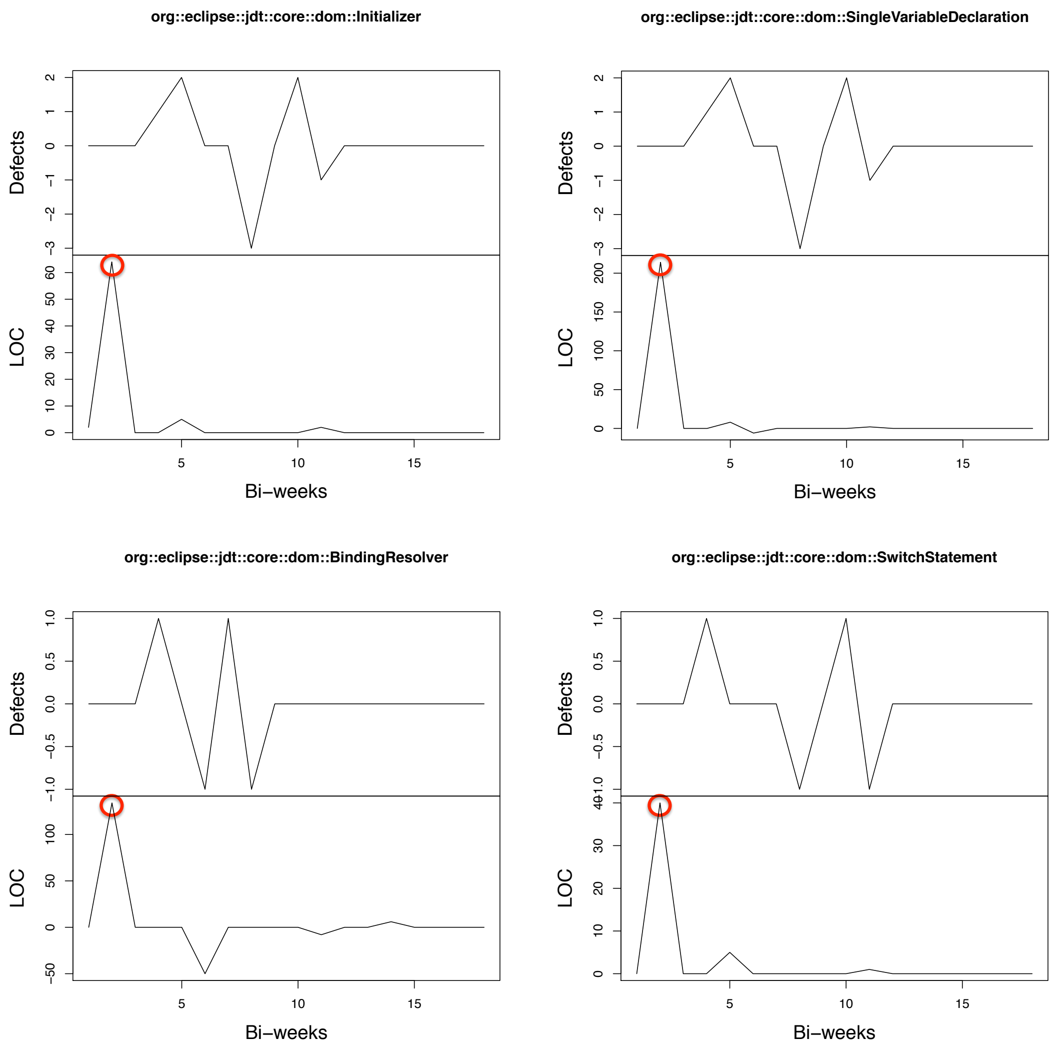


Figure 6.7: True alarms raised by our approach

recall were 68%, 44%, 31%, and 52%, respectively.

*RQ1: Our approach reached an average precision greater than 50% in three out of the four systems we evaluated. On the other hand, as expected, we were not able to trigger alarms for all defects using times series of source code metrics as predictors. On average, we achieved recall rates ranging from 13% (Equinox Framework) to 31% (Lucene).*

#### 6.4.2.2 RQ2: How does our approach compares with the proposed baselines?

This research question aims to compare the precision of our approach with three baselines. Figure 6.8 shows for each time frame the precision results for the following models: (a) proposed approach (*Granger*); (b) *Baseline1* (baseline that does not consider the results of the Granger Test); (c) *Linear* (baseline that uses simple linear regression as the prediction technique). As we can note, the initial time frames have no precision results. This lack of precision happened because we discarded results coming from unstable models, i.e., models reporting zero alarms or whose precision values alternate between 0 and 1. As we can observe in the figure, in most time frames, our approach (solid line) shows a precision greater than *Baseline1* (long dash line) and *Linear* (dotted line). To confirm this assumption, for each pair of samples (*Granger* vs. *Baseline1* and *Granger* vs. *Linear*), we applied a paired test for means (Student's $t$-test) using a significance level of 95%. This test confirmed that the mean precision of our approach (*Granger*) is significantly different from *Baseline1* in three out of the four systems (Eclipse PDE UI, Equinox, and Lucene) . Furthermore, the mean precision of our approach is also significantly different from *Linear* in three out of the four systems (Eclipse JDT Core, Eclipse PDE UI, and Equinox).

It is worth mentioning that Figure 6.8 also shows that in several time frames our approach reached high precision measures. For instance, for Eclipse JDT Core, between time frames 36 and 47, our models achieved a precision ranging from 83% to 90%, with the number of true alarms ranging from 40 to 138. For Eclipse PDE UI, our approach in the time frame 24 reached a precision of 60%, with three true alarms. For Equinox, between time frames 95 and 107, our approach reached a precision ranging from 60% to 75%, with the number of true alarms ranging from 6 to 11. Finally, for Lucene, between time frames 79 and 88, our approach reached a precision ranging from 66% to 87%, with the number of true alarms ranging from 2 to 7.

Figure 6.9 shows for each time frame the precision results for the following models: (a) *Baseline1* and; (b) *Baseline2* (baseline that only considers precondition P1). As we can observe, in most time frames, *Baseline1* (solid line) shows a precision greater than *Baseline2* (long dash line). In fact, the $t$-test asserted that the mean precision of *Baseline1* is significantly different from *Baseline2* (for this reason, we omitted *Baseline2* from Figure 6.8).

It is also important to highlight that the precision results do not present a monotonically increasing behavior, as the evaluated models include more bi-weeks in the respective training time series. For example, the highest precision value for Eclipse JDT Core was achieved in bi-week 42 (precision= 90%) and the second lowest value

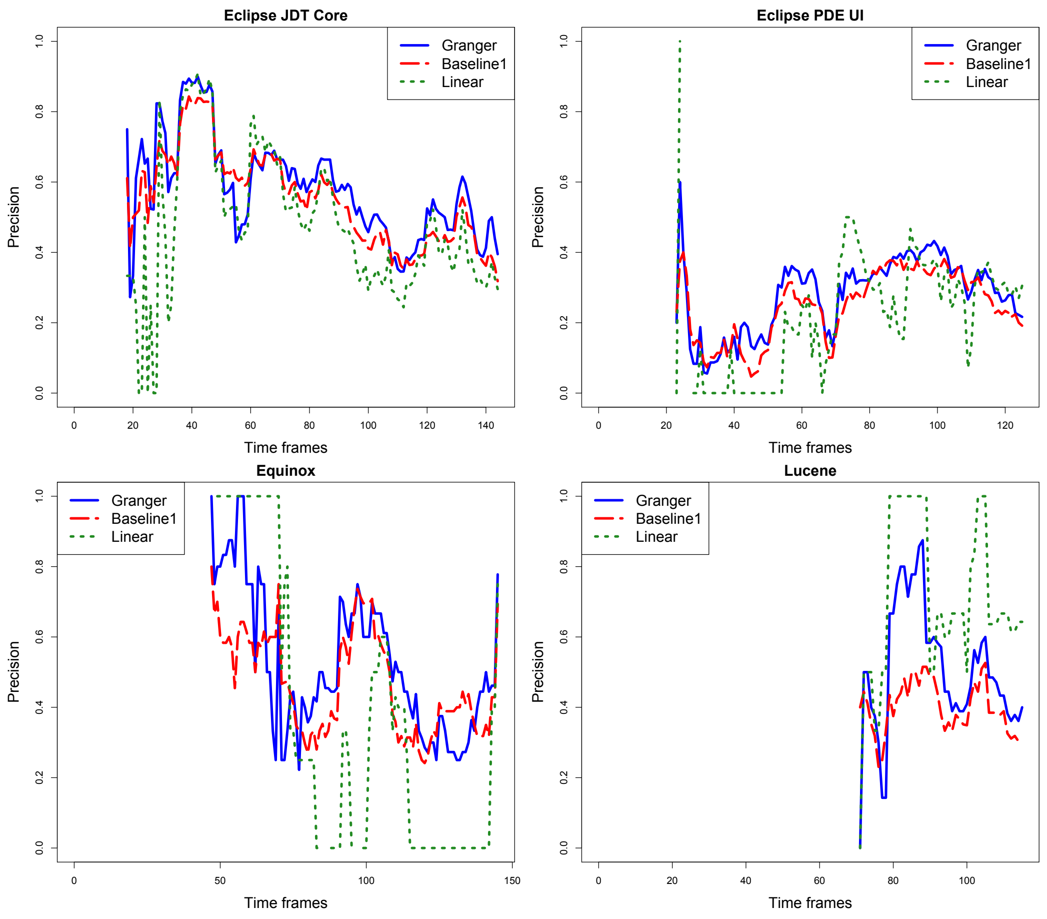


Figure 6.8: Precision results

69 bi-weeks later (precision= 34%).

Considering recall, Figure 6.10 compares our approach with two baselines: *Baseline1* and *Linear*. In general terms, the model based on Granger outperformed the baseline that uses simple linear regression as the prediction technique (*Linear*). However, the best recall was achieved by the baseline that does not consider the results of Granger or of any other prediction technique (*Baseline1*). In fact, this result is expected since *Baseline1* triggers more alarms and therefore such alarms have more chances to cover real defects.

To summarize, two conclusions can be derived from our investigation concerned this research question: (i) based on the fact that the *Baseline1* outperformed the *Baseline2*, we can conclude that when building defect prediction models, it is important to remove classes with zero defects (that make the predictions trivial), classes with a constant behavior (that do not contribute with predictive power), and classes with non-stationary time series (that may statistically invalidate the findings); and (ii) from

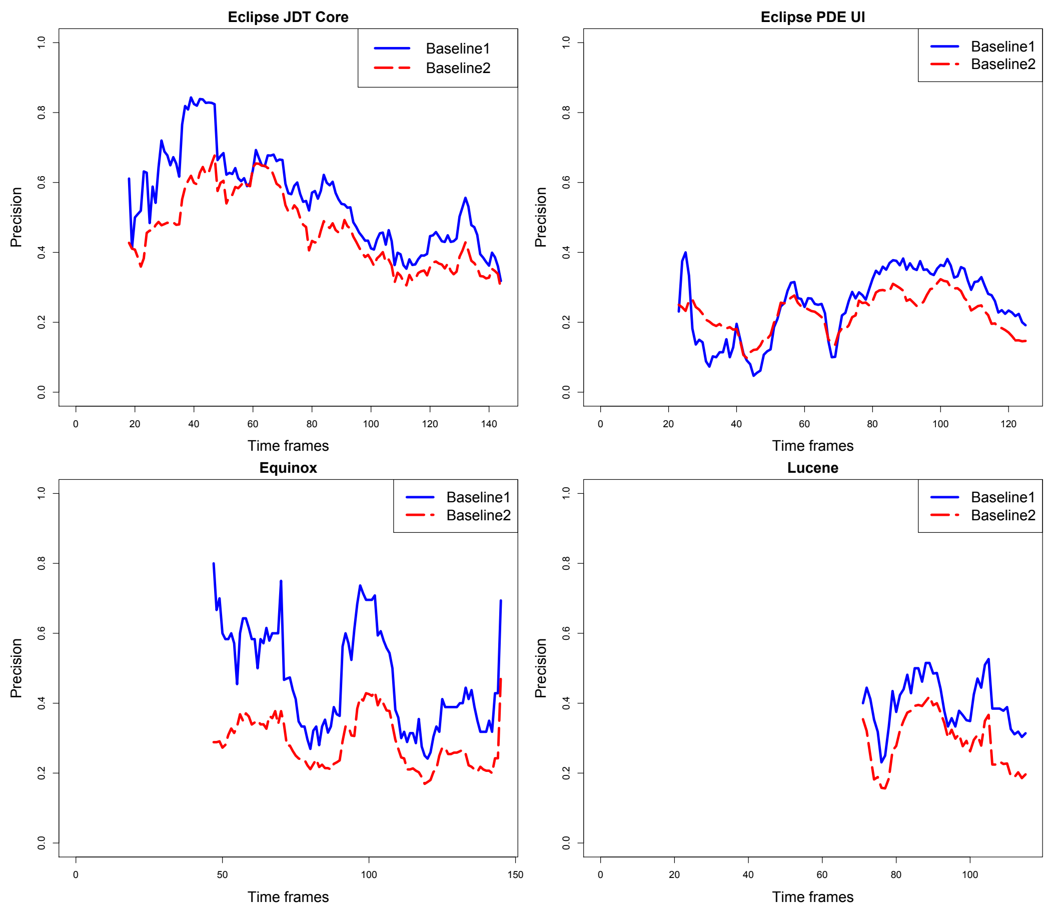


Figure 6.9: Precision results of *Baseline1* and *Baseline2*

the fact that *Granger* outperformed both *Baseline1* and *Linear*, we can conclude that it is possible to achieve gains in precision by considering the Granger Test to raise alarms for defects (instead of traditional models, based for example on standard regressions).

*RQ2: The precision achieved by our approach was statistically better than the proposed baselines in three out of the four systems, which confirms the gains achieved by considering Granger-causality when predicting defects using source code metrics.*

#### 6.4.2.3 RQ3: What is the impact in the precision results of using other functions (different from the mean) for triggering alarms?

To answer this third research question, the experiments conducted for answering the first questions were executed again changing only the function used to summarize the variations in the values of the source code metrics. More specifically, in the previous

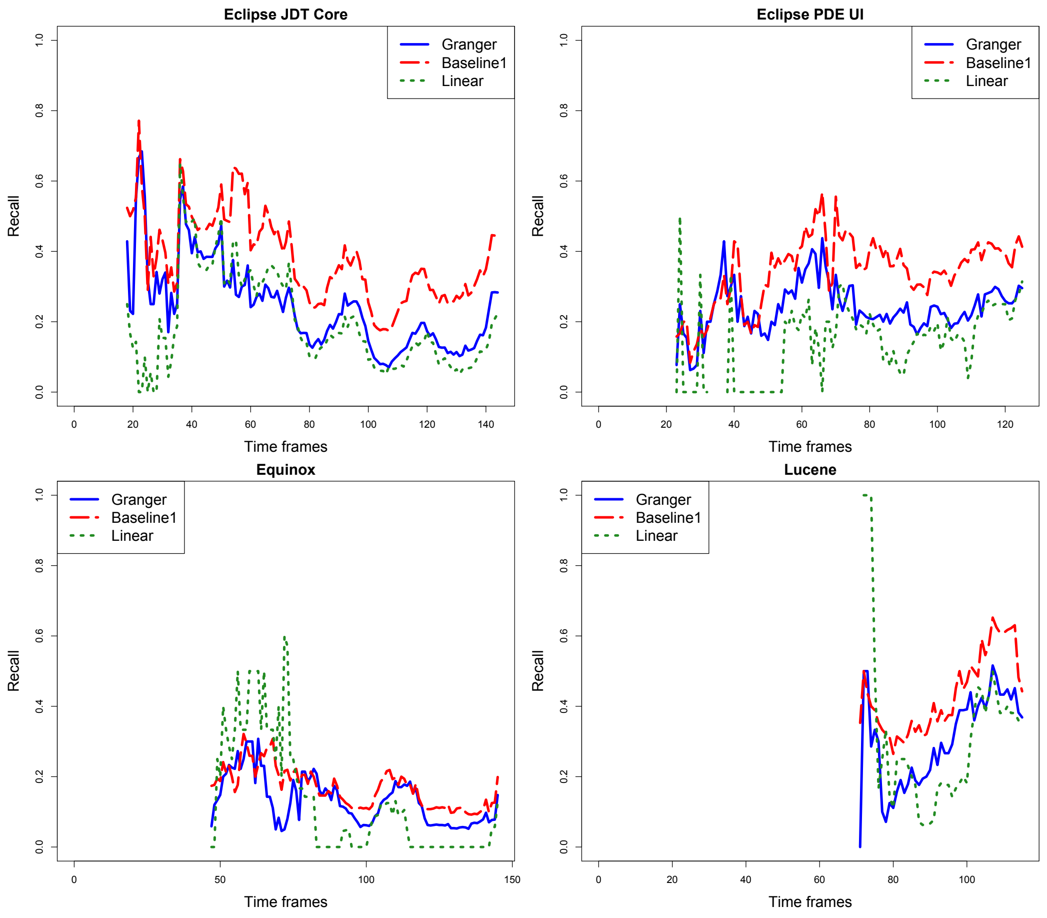


Figure 6.10: Recall results

research questions the thresholds used to trigger alarms were defined as the *mean* of the positive variations in the values of the source code metrics that Granger-caused defects in a given class. On the other hand, in this research question, we evaluated five other descriptive statistics functions calculated over such variations: minimal value, first quartile value, median, third quartile value, and maximal value. For example, suppose that during the training window it was inferred that variations in the values of a given metric $m$ Granger-caused defects in a class $C$. Therefore, a model based on the *minimal* function will trigger alarms for this class, during the validation window, when the value of $m$ after a given change in $C$ is greater than the *minimal* variation of $m$ observed during the training period.

Table 6.2 presents the average precision and recall achieved by the evaluated models considering the aforementioned descriptive statistics functions. As can be observed, the precision is not affected by the considered functions. For example, for the Eclipse JDT Core system, the precision ranges from 51% (for alarms thresholds calculated using the maximal variation in the source code metrics values) to 61% (for thresholds equal to the minimal variation values). In the other systems, the dispersion was less than this one reported for the Eclipse JDT Core. Regarding recall, the best result is always achieved by the *minimal* function, since this function raises more alarms. Conversely, the worst recall is achieved by the *maximal* function.

Figure 6.11 complements this finding by showing the number of alarms (Figure 6.11a) and the number of true alarms (Figure 6.11b) for each model evaluated in our study, considering only the Eclipse JDT Core. As expected, by changing the functions from the minimal to the maximal functions, the number of alarms decreases, i.e., the higher the threshold, the lower the number of alarms. However, Figure 6.11b shows that the true alarms change in the same proportion, as we change the number of alarms. For this reason, the precision among the different threshold functions was very similar.

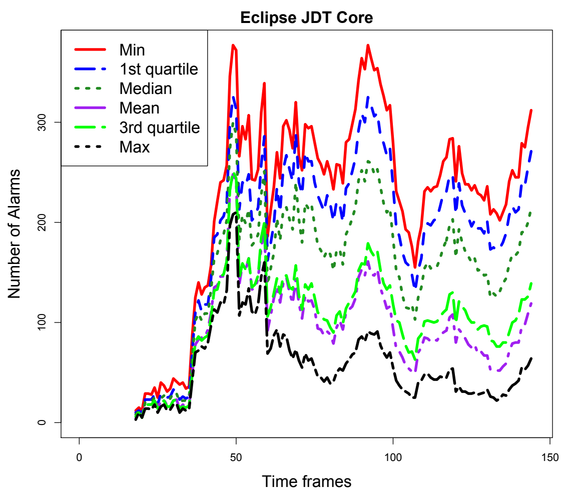
Table 6.2: Average precision (Pre) and recall (Rec) results considering alternative threshold functions

| **Function** | **JDT Core** | | **PDE UI** | | **Equinox** | | **Lucene** | |
|---|---|---|---|---|---|---|---|---|
| | **Pre** | **Rec** | **Pre** | **Rec** | **Pre** | **Rec** | **Pre** | **Rec** |
| Min | 61% | 53% | 27% | 54% | 49% | 33% | 51% | 51% |
| 1st Quartile | 60% | 45% | 27% | 50% | 50% | 30% | 52% | 40% |
| Median | 59% | 37% | 27% | 43% | 49% | 24% | 53% | 36% |
| Mean | 58% | 24% | 28% | 24% | 52% | 13% | 51% | 31% |
| 3rd Quartile | 57% | 26% | 25% | 29% | 51% | 17% | 50% | 29% |
| Max | 51% | 15% | 28% | 16% | 51% | 10% | 50% | 20% |

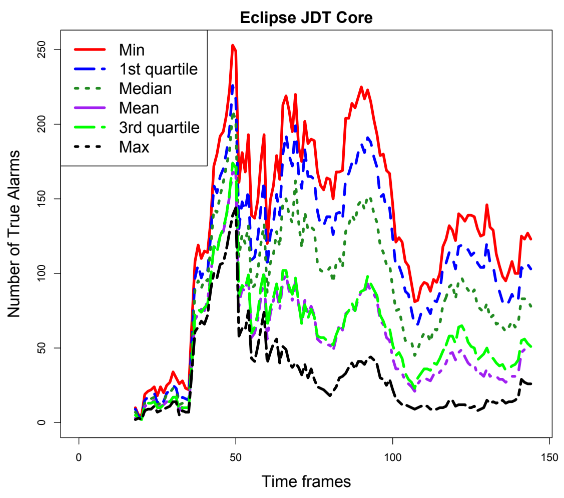
*RQ3: Because the evaluated functions presented similar results, we decided to trigger alarms using the mean variation in the values of the source code metrics, as originally proposed.*

#### 6.4.2.4 RQ4: Regarding their severity, what types of bugs are typically predicted by the proposed models?

To answer the fourth research question, we followed these steps: (a) for each alarm classified as a true alarm in our experiments, we located the defect responsible for this classification (called predicted defect); (b) each predicted defect was then mapped to its respective bug (called predicted bug), following a reverse process from the one described in Section 4.2.2; (c) for each predicted bug, we retrieved its severity degree



(a) Alarms



(b) True Alarms

Figure 6.11: Number of alarms and true alarms, for different threshold functions (Eclipse JDT Core)

in the Jira and Bugzilla tracking platforms. Basically, when reporting a bug, an user of such platforms can rank the bug in one of the following categories: blocker, critical, major, normal, minor, or trivial.

Figure 6.12 presents the distribution of the bugs considered in our experiment by severity. The figure shows the distribution of the whole population of bugs considered in the study and also the distribution of the bugs predicted by at least one of our models. As we can observe, the distributions are very similar, in all systems in our dataset. For example, 81% of the bugs we evaluated in the Eclipse JDT Core system are normal bugs; for this system, 84% of the bugs predicted by our approach were also classified as normal, regarding their severity.

*RQ4: In terms of severity, we were not able to identify a particular category of bugs that the proposed model tends to predict with higher frequency.*

## 6.5 Threats to Validity

In this section, we discuss potential threats to the validity of our study. We arranged possible threats in three categories: external, internal, and construct validity [PPV97]:

| **Severity** | **Bugs** | | **Predicted** | |
|---|---|---|---|---|
| | # | % | # | % |
| Blocker | 38 | 1 | 11 | 1 |
| Critical | 131 | 4 | 24 | 3 |
| Major | 311 | 8 | 61 | 7 |
| Normal | 3001 | 81 | 694 | 84 |
| Minor | 156 | 4 | 33 | 4 |
| Trivial | 60 | 2 | 4 | 0 |
| Total | 3697 | 100 | 827 | 100 |

(a) Eclipse JDT Core

| **Severity** | **Bugs** | | **Predicted** | |
|---|---|---|---|---|
| | # | % | # | % |
| Blocker | 14 | 1 | 1 | 1 |
| Critical | 52 | 3 | 4 | 2 |
| Major | 115 | 6 | 9 | 5 |
| Minor | 77 | 4 | 5 | 3 |
| Normal | 1509 | 84 | 159 | 89 |
| Trivial | 31 | 2 | 1 | 1 |
| Total | 1798 | 100 | 179 | 100 |

(b) Eclipse PDE UI

| **Severity** | **Bugs** | | **Predicted** | |
|---|---|---|---|---|
| | # | % | # | % |
| Blocker | 14 | 2 | 1 | 2 |
| Critical | 33 | 4 | 1 | 2 |
| Major | 55 | 7 | 4 | 7 |
| Minor | 12 | 2 | - | - |
| Normal | 661 | 84 | 48 | 89 |
| Trivial | 9 | 1 | - | - |
| Total | 784 | 100 | 54 | 100 |

(c) Equinox Framework

| **Severity** | **Bugs** | | **Predicted** | |
|---|---|---|---|---|
| | # | % | # | % |
| Blocker | 3 | 1 | - | - |
| Critical | 6 | 2 | - | - |
| Major | 143 | 43 | 20 | 42 |
| Minor | 160 | 48 | 27 | 56 |
| Normal | - | - | - | - |
| Trivial | 23 | 7 | 1 | 2 |
| Total | 335 | 100 | 48 | 100 |

(d) Lucene

Figure 6.12: Distribution of bugs by severity

*External Validity:* Our study to evaluate the proposed defect prediction model involved four medium-to-large systems, including three systems from the Eclipse project and one system from the Apache Foundation, with a total of 6,223 classes. Therefore, we claim this sample includes a credible number of classes, extracted from real-world and non-trivial applications, with a consolidated number of users and a relevant history of bugs. Moreover, we considered seven metrics, covering major source code properties like size, coupling, and cohesion. Despite these observations, we can not guarantee—as usual in empirical software engineering—that our findings apply to other metrics or systems, specifically to systems implemented in other languages or to systems from different domains, such as real-time systems and embedded systems.

*Internal Validity:* This form of validity concerns to internal factors that may influence our observations. A possible internal validity threat concerns the procedure to identify the thresholds used by our model to trigger alarms. We rely on the average of the positive variations in the metric values to define such thresholds. We acknowledge that the use of the average in this case is not strictly recommended, because we never checked whether the positive variations follow a normal distribution. However, we tested other functions (median, 1st quartile, minimum, etc.) and they presented similar results than the average.

*Construct Validity:* This form of validity concerns the relationship between theory and observation. A possible threat concerns the way we linked bugs to defects in classes. Particularly, we discarded bugs without explicit references in the textual description of the commits. However, the percentage of such bugs was not large, around 36% of the bugs considered in our evaluation. Moreover, this approach is commonly used in studies that involve mapping bugs to defects in classes [DLR10, ZPZ07, SZZ05b].

## 6.6 Final Remarks

In this chapter, we proposed and evaluated an approach for predicting defects using causality tests. Our approach reached an average precision greater than 50% considering three out of the four systems we evaluated. Moreover, the results showed that the precision of the alarms changes with time. For example, for the Eclipse JDT Core, we achieved an average precision of 58% considering 144 models covering seven years of the system's history, and including a minimal and maximal precision of 27% and 90%, respectively. On the other hand, we were not able to predict all defects using times series of source code metrics. On average, we achieved recall rates ranging from 13% (Equinox Framework) to 31% (Lucene). In fact, we argued that it is not feasible to expect that alarms based on source code metrics can cover the whole spectrum of bugs reported to a system. Finally, we showed that our models outperform models that trigger alarms without considering Granger-causality or that are based on linear regression techniques.

# Chapter 7

# BugMaps-Granger

In this chapter, we start by presenting BugMaps, a tool for the visual exploration and analysis of bugs (Section 7.3). In Section 7.3, we introduce the BugMaps-Granger tool—an extension of BugMaps—that supports detection of causal relations between source code metrics and bugs. In Section 7.4, we present a case study, where we used BugMaps-Granger to discover causal relations between metrics values and bugs in two systems: Equinox Framework and Eclipse JDT Core systems. Finally, we discuss related tools (Section 7.5).

## 7.1 Motivation

A number of software analysis tools has been proposed recently to improve software quality [NDG05a, HP04, Son13, Wet09]. Such tools use different types of information about the structure and history of software systems. Basically, they can be used to analyze software evolution, manage the quality of the source code, compute metrics, check coding rules, etc. In general, such tools help maintainers to understand large amounts of data coming from software repositories.

Particularly, there is a growing interest in software analysis tools for exploring bugs in software systems [HCA$^+$12, DL07, DL10, SZZ05a]. Such tools help maintainers understand the distribution, the evolutionary behavior, the lifetime, and the stability of bugs. Basically, they work by retrieving history data from bug tracking and version control platforms, by mapping bugs to defects in source code modules, and by processing data to extract and reason about bugs and defects.

Despite the increasing number of bug analysis tools, they typically do not provide mechanisms for assessing the existence of correlations between the internal quality of a software system and the occurrence of bugs. To the best of our knowledge, there are no bug analysis tools that the highlight possible causes of bugs in the source code of a system. More specifically, there are no tools designed to infer eventual causal relations between source code properties (as measured by source code metrics) and the occurrence of defects in object-oriented classes.

In this chapter, we propose and describe the BugMaps-Granger tool—an extension of the BugMaps tool [HCA$^+$12]—that supports detection of causal relations between source code metrics and bugs. The tool provides mechanisms to retrieve data from software repositories, to compute source code metrics, to generate time series of source code metrics and defects, and to infer causal relations between source code properties and defects. Moreover, BugMaps-Granger provides visualizations for identifying the modules with more bugs, the average lifetime and complexity of bugs, and the source cod properties that are more likely to cause bugs. More specifically, our tool relies on the Granger Causality Test [Gra81] to identify causal relations between time series of source code metrics and defects. As described in Chapter 3, this test evaluates whether past changes to a given time series of metrics can be used to forecast changes in a time series of defects. The proposed tool has the following features:

- The tool automatically extracts source code models of a target system from its version control platform in predefined time intervals.
- The tool generates time series of twelve source code metrics and time series with the number of defects in each class of the target system.
- The tool applies the Granger Test considering the metrics and defects time series to highlight possible causal relations.
- The tool integrates models extracted from the source code with models representing the number of bugs.
- The tool provides a set of interactive visualizations to support software maintainers in answering questions such as: (a) Which are the modules with more bugs? (b) What is the average lifetime of bugs? (c) What is the complexity of bugs? (d) What are the source code properties that Granger-cause bugs in a given module?, and (e) What are the metrics with the highest number of positive Granger tests?

The ultimate goal of BugMaps-Granger is to predict the changes in the source code of a software that are more likely to cause defects. For example, with our tool in hand, a maintainer (before making a commit with changes to a given class) can verify whether such changes significantly affect the values of source code metrics that, in the past, Granger-caused defects. If the changes affect these metrics values, the maintainer can, for example, perform extra software quality assurance activities in this class (e.g., she can conduct more unit testing or perform a detailed code inspection) before executing the commit.

## 7.2 BugMaps

BugMaps is a tool for the visual exploration and analysis of bugs. This tool was developed in partnership with INRIA-France [HCA+12]. Basically, this tool provides mechanisms to automate the process of retrieving and parsing software repositories data, and algorithms to map bugs reported in bug tracking platforms to defects in the classes of object-oriented systems. It also provides visualizations for decision support.

Figure 7.1 shows the BugMaps architecture, which includes the following modules:

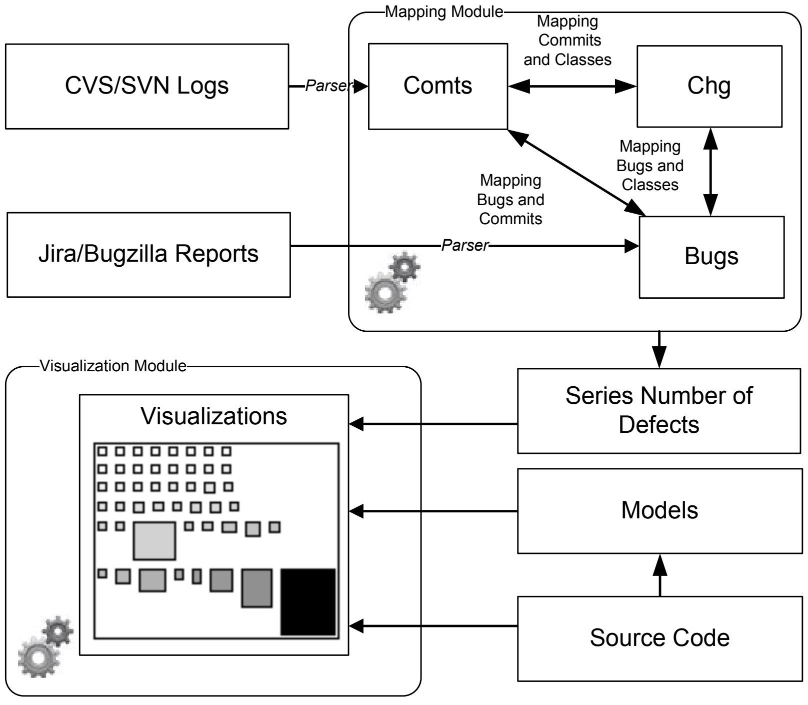


Figure 7.1: BugMaps architecture

**Mapping Module**: This module receives as input the log files from version control platforms—CVS or SVN—and the bug reports from bug tracking platforms—Jira or Bugzilla. Basically, this module maps bugs to defects in classes and creates the times series with the number of defects (i.e., for each class, a time series that provides the number of defects in a given time frame). To create this time series, we implemented an XML parser that reads the information provided by the CVS/SVN repositories and extracts the developer's comments and the changed classes. Then, another XML parser reads the bug reports available in the Jira/Bugzilla repositories and collects the date on which each bug was reported and its identifier. After that, we linked each bug $b$ to the classes changed to fix $b$, using the algorithm described in Section 4.2.2.

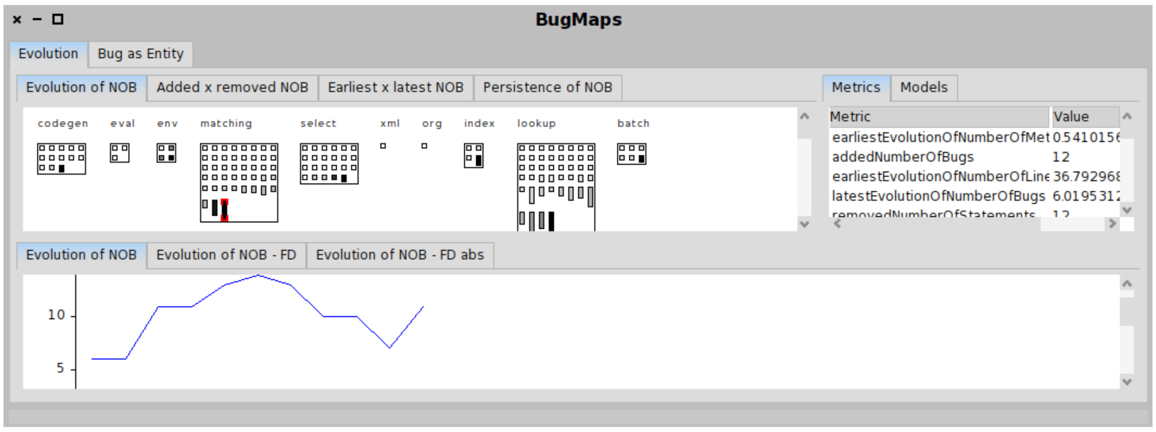
**Visualization Module:** This module receives as input the series number of defects, the meta-models extracted from several versions of the source code and the source code itself. These meta-models are generated using VerveineJ parser[1]. Two browsers are then used for analysis, one that deals with the history of bugs (called history browser, which receives as input a history model [GD06]) and another that deals with a particular snapshot of the system under analysis (called snapshot browser, which receives as input a snapshot model). These browsers are implemented in the Moose Platform[2]. Moose is a platform that provides an infrastructure for building reverse and re-engineering software analysis tools [NDG05b].

The visualizations provided by BugMaps are based on Distribution Map, a generic technique to reason about the results of software analysis and to display how a given phenomenon is distributed across a software system [DGK06]. Using a Distribution Map, three metrics can be displayed through the height, width, and color of the objects in the map. In our maps, rectangles represent classes or bugs and containers represent packages or years.

Figure 7.2 shows the history browser, which is composed by three panes: visualizations (top left), metrics (top right) and charts (bottom). The *charts* pane shows the number of bugs presented in a class/package during its lifetime and the *metrics* pane shows class/package measures, which are updated according to the selected entity in the *visualizations* pane. The visualizations can be swapped using tabs presented in the top of the *visualizations* pane.

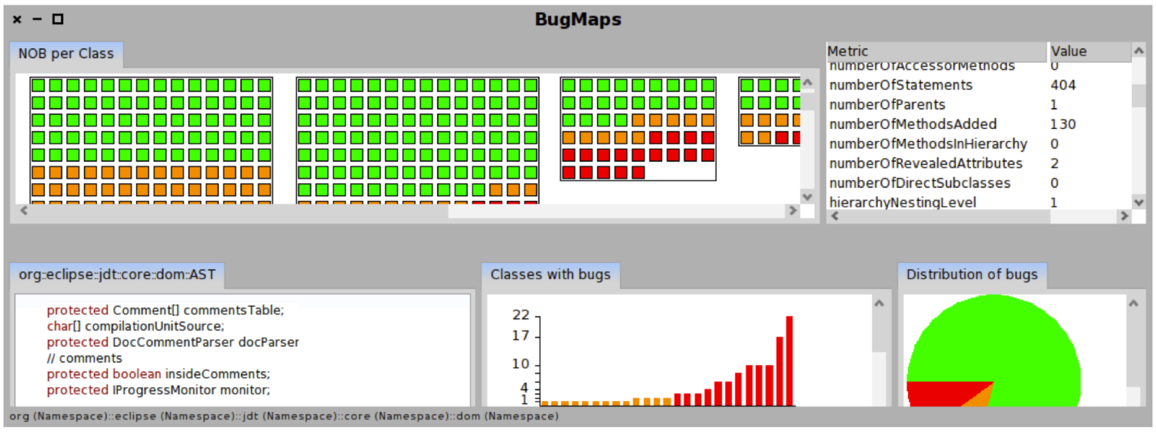
From the history browser (a collection of snapshot models), it is possible to open snapshot browsers using tabs presented in the top of the *metrics* pane. Figure 7.3 shows the snapshot browser, which is composed by four panes: visualizations (top left), metrics (top right), source code (bottom left) and charts (bottom right). *Metrics*, *source code* and *charts* panes are also updated according to the selected entity in the *visualizations* pane. BugMaps is publicly available for download at:

[1]http://www.moosetechnology.org/tools/verveinej
[2]http://www.moosetechnology.org



| Metric | Value |
|---|---|
| earliestEvolutionOfNumberOfMet | 0.5410156 |
| addedNumberOfBugs | 12 |
| earliestEvolutionOfNumberOfLine | 36.792968 |
| latestEvolutionOfNumberOfBugs | 6.0195312 |
| removedNumberOfStatements | 12 |

Figure 7.2: History Browser



| Metric | Value |
|---|---|
| numberOfAccessorMethods | 0 |
| numberOfStatements | 404 |
| numberOfParents | 1 |
| numberOfMethodsAdded | 130 |
| numberOfMethodsInHierarchy | 0 |
| numberOfRevealedAttributes | 2 |
| numberOfDirectSubclasses | 0 |
| hierarchyNestingLevel | 1 |

org::eclipse::jdt::core::dom::AST

```
protected Comment[] commentsTable;
char[] compilationUnitSource;
protected DocCommentParser docParser
// comments
protected boolean insideComments;
protected IProgressMonitor monitor;
```

org (Namespace)::eclipse (Namespace)::jdt (Namespace)::core (Namespace)::dom (Namespace)

Figure 7.3: Snapshot Browser

http://rmod.lille.inria.fr/web/pier/software/BugMaps

## 7.3 BugMaps-Granger

BugMaps-Granger is an extension of the BugMaps tool that allows the analysis of source code properties that Granger-caused bugs. This tool automatically extracts source code versions from version control platforms, generates source code metrics and defects time series, computes Granger Causality Test, and provides interactive visualizations for causal analysis of bugs.

The execution of the BugMaps-Granger tool is divided into two phases: preprocessing and visualization. The preprocessing phase is responsible for extracting source code models, creating time series, and applying the Granger Test to compute possible causal relations between source code metrics and bugs. In the visualization phase, the user can interact with the tool. For example, he can retrieve the most defective classes of the system and visualize the source code properties that Granger-caused bugs in these classes.

The following data are required to execute the tool: (i) identifiers and creation dates of bugs, stored in a CSV file; and (ii) URL of the version control platform (SVN or GIT) containing the source code of the target system. It is also important to highlight that bugs reported in the bug tracking platforms must be collected during the same time frame used to extract the source code versions.

### 7.3.1 Architecture

Figure 7.4 shows the architecture of the BugMaps-Granger, which includes four modules: model extraction, time series creation, Granger Test module, and visualization module.

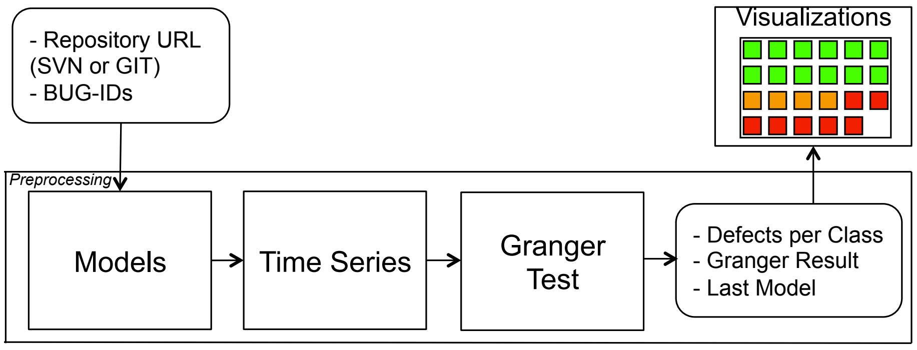


Figure 7.4: BugMaps-Granger's architecture

In the following paragraphs, we describe the modules of this architecture:

**Model Extraction:** This module receives as input the URL associated to the version control platform of the target system (SVN or Git) and a time interval to be used in the analysis of the bugs. To extract the source code models, the module performs the following tasks: (a) it extracts the source code versions from the version control platforms in intervals of bi-weeks; (b) it removes test classes, assuming that such classes are implemented in directories and subdirectories whose name starts with the words "Test" or "test"; and (c) it parses the source code versions and generates MSE [DAB+11] files using the VerveineJ tool[3]. MSE is the default file format supported by the Moose platform to persist source code models.

[3]http://www.moosetechnology.org/tools/verveinej.

**Time Series Creation:** To create the time series of source code metrics, this module receives as input the meta-models extracted by the previous module. For each class of each extracted model, the module relies again on the Moose platform to compute eleven source code metrics including six CK metrics (proposed by Chidamber and Kemerer [CK94]) and five others, such as lines of code, FAN-IN, FAN-OUT, etc. Table 7.1 shows the source code metrics considered by the tool. To create the time series of defects for each class, this module receives as input a file containing the BUG-IDs and the bug creation dates collected from the bug tracking platforms of the target system. Basically, this module maps the bugs to their respective commits using the BUG-ID. Next, the files changed by such commits are used to identify the classes changed to fix the respective bugs. More details about the creation of the time series of source code metrics and defects can be checked in Section 4.2.1 and Section 4.2.2, respectively.

Table 7.1: Source code metrics considered by BugMaps-Granger

| | **Metrics** | **Description** |
|---|---|---|
| 1 | WMC | Weighted methods per class |
| 2 | DIT | Depth of inheritance tree |
| 3 | RFC | Request for class |
| 4 | NOC | Number of children |
| 5 | CBO | Coupling between object class |
| 6 | LCOM | Lack of cohesion in methods |
| 7 | FAN-IN | Number of classes that reference a given class |
| 8 | FAN-OUT | Number of classes referenced by a given class |
| 9 | NOA | Number of attributes |
| 10 | LOC | Number of lines of code |
| 11 | NOM | Number of methods |

**Granger Test Module:** This module applies the Granger Causality Test considering the metrics and defects time series. More specifically, Granger is responsible for identifying causal relations between time series of source code metrics and defects. To apply the test, this module relies on the function *granger.test()* provided by the *msbvar* package from the R statistical system. More details about the Granger Test and the algorithm used to identify causal relations can be checked in Section 3.3.2.

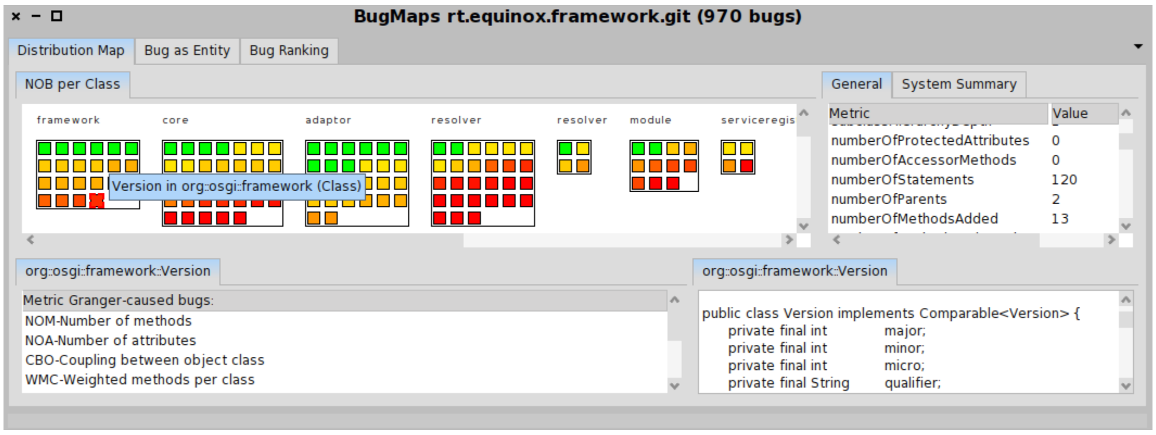
**Visualization Module:** This module receives the following input data: a file containing the bugs mapped to their respective classes and the Granger results, a model extracted from the last source code version, and the source code itself of the system under analysis. From this information, the module provides four interactive visualization browsers:

- Two browsers are used for analysis. The first one deals with the classes, the number of bugs, and the Granger results of the system under analysis (called Granger browser) while the second one deals with the complexity of the bugs (called Bug as Entity browser).

- Two browsers are used to to rank the classes and the metrics most involved with bugs.

Such browsers are implemented using visualization packages provided by the Moose Platform. Basically, the visualizations are based on Distribution Map, a generic technique to reason about the results of software analysis and to investigate how a given phenomenon is distributed across a software system [DGK06]. Using a Distribution Map three metrics can be displayed through the height, width, and color of the objects in the map. In our maps, rectangles represent classes or bugs and containers represent packages.

Figure 7.5 shows the Granger browser, which has four panes: visualization of classes and packages (top left), measures (top right), Granger results (bottom left), and source code (bottom right)[4]. Metrics, source code, and Granger results are updated according to the selected class in the classes and packages pane.


BugMaps rt.equinox.framework.git (970 bugs)
Distribution Map
Bug as Entity
Bug Ranking
NOB per Class
framework
core
adaptor
resolver
resolver
module
serviceregis
Version in org::osgi::framework (Class)
General
System Summary
org::osgi::framework::Version
org::osgi::framework::Version

| Metric | Value |
|---|---|
| numberOfProtectedAttributes | 0 |
| numberOfAccessorMethods | 0 |
| numberOfStatements | 120 |
| numberOfParents | 2 |
| numberOfMethodsAdded | 13 |

Metric Granger-caused bugs:
NOM-Number of methods
NOA-Number of attributes
CBO-Coupling between object class
WMC-Weighted methods per class

```
public class Version implements Comparable<Version> {
    private final int          major;
    private final int          minor;
    private final int          micro;
    private final String       qualifier;
```

Figure 7.5: Granger browser

Figure 7.6 shows the Bug as Entity browser which is composed by two panes: visualization of classes and packages (left pane) and bugs (right pane). When a defective class is selected, the bugs in the class are colored in black (in the right pane).

[4]Since most of our visualizations make heavy use of colors, we provide high-resolution versions of these figures in a companion website: http://aserg.labsoft.dcc.ufmg.br/bugmaps.

In contrast, when a bug is selected, the classes changed to fix this bug are colored in black (in the left pane). BugMaps-Granger also shows a list of classes ranked by the number of defects and the number of Granger tests that returned a positive result.

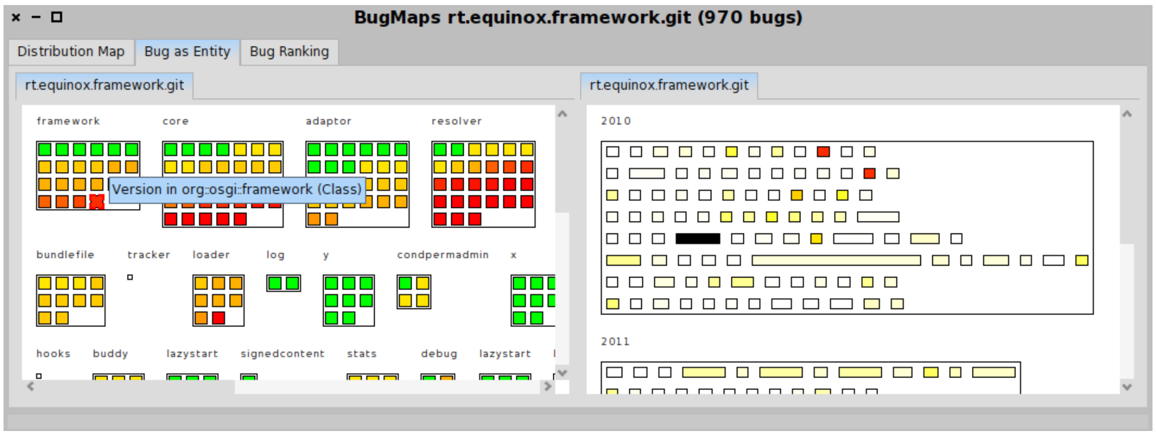


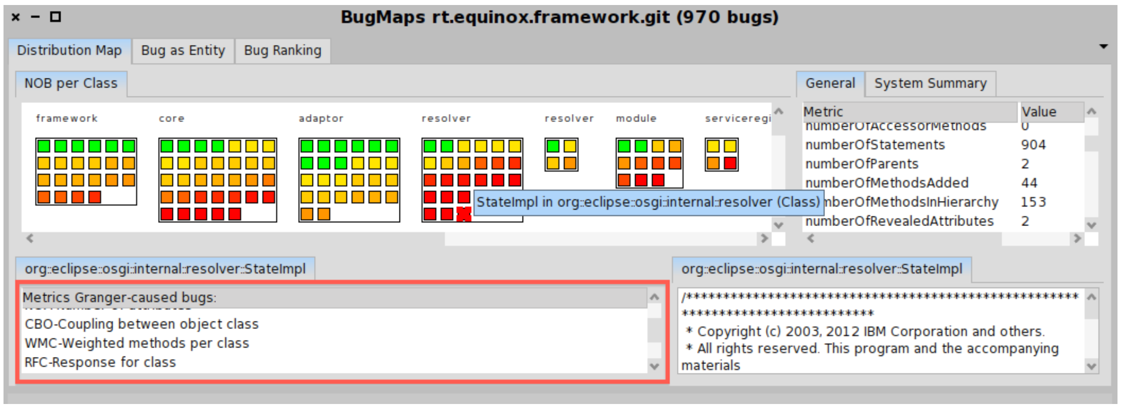
Figure 7.6: Bug as Entity browser

## 7.4 Case Study

In this section, we provide an example of use considering data from the Equinox Framework and Eclipse JDT Core systems collected during three years. For Equinox Framework, the tool extracted 79 source code versions in intervals of bi-weeks, including 417 classes, from 2010-01-01 to 2012-12-28. For Eclipse JDT Core, the tool extracted 78 source code versions in intervals of bi-weeks, including 2,467 classes, from 2005-01-01 to 2007-12-15. In a second step, for each class, the tool created eleven time series of source code metrics (one for each metric in Table 7.1) and one time series of defects. Finally, for each pair of time series (source code metrics and defects), the tool applied the Granger Test to identify causal relations. We analyzed the Granger results for both systems according to the proposed visualizations, as discussed next.

**Granger:** In this map, the rectangles are the classes of the target system, as illustrated in Figure 7.7. The color of a class represents the number of bugs detected through its history ranging from green to red (the closer to red, more bugs the class had in its history). By selecting a defective class, the bottom pane is updated showing the source code metrics that Granger-caused bugs in this class. Particularly, Figure 7.7 provides an overview of the distribution of the bugs in the Equinox Framework system. We can observe that the `resolver` package contains a significant number of classes with bugs. Moreover, for the class `org.eclipse.osgi.internal.resolver.StateImpl`, we can observe that the source code metrics that Granger-caused bugs were CBO (Coupling Between Object), WMC (Weighted Methods per Class), and RFC (Response for Class).

The Granger browser can be used to avoid future defects. For example, with this result in hand, a maintainer (before making a commit with changes to the `StateImpl` class) can verify whether such changes heavily affect the values of source code metrics that Granger-caused defects in the past (in our example, the metrics that Granger-caused defects were CBO, WMC, and RFC). If the change affects these metrics, the maintainer can for example perform extra software quality assurance activities in this class (like unit testing or code inspection).


BugMaps rt.equinox.framework.git (970 bugs)
Distribution Map
Bug as Entity
Bug Ranking
NOB per Class
framework
core
adaptor
resolver
resolver
module
serviceregi
StateImpl in org::eclipse::osgi::internal::resolver (Class)
General
System Summary
org::eclipse::osgi::internal::resolver::StateImpl
org::eclipse::osgi::internal::resolver::StateImpl

| Metric | Value |
|---|---|
| numberOfAccessorMethods | 0 |
| numberOfStatements | 904 |
| numberOfParents | 2 |
| numberOfMethodsAdded | 44 |
| numberOfMethodsInHierarchy | 153 |
| numberOfRevealedAttributes | 2 |

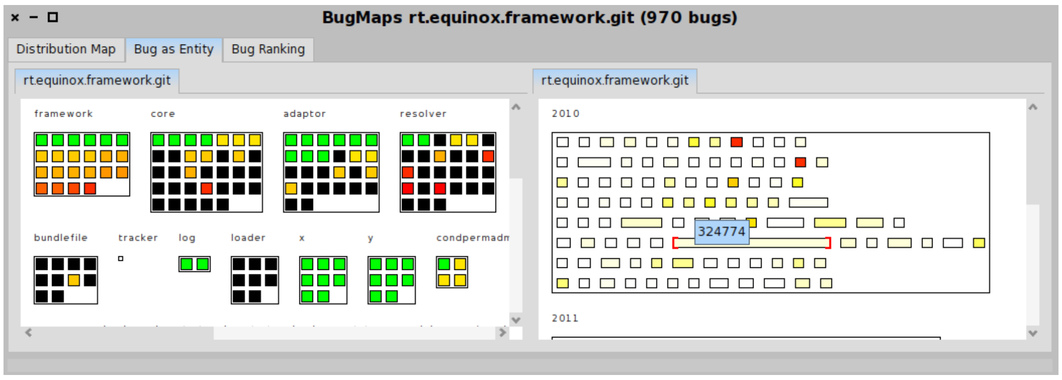
Metrics Granger-caused bugs:
CBO-Coupling between object class
WMC-Weighted methods per class
RFC-Response for class

```
/*****************************************************
*************************
 * Copyright (c) 2003, 2012 IBM Corporation and others.
 * All rights reserved. This program and the accompanying
materials
```

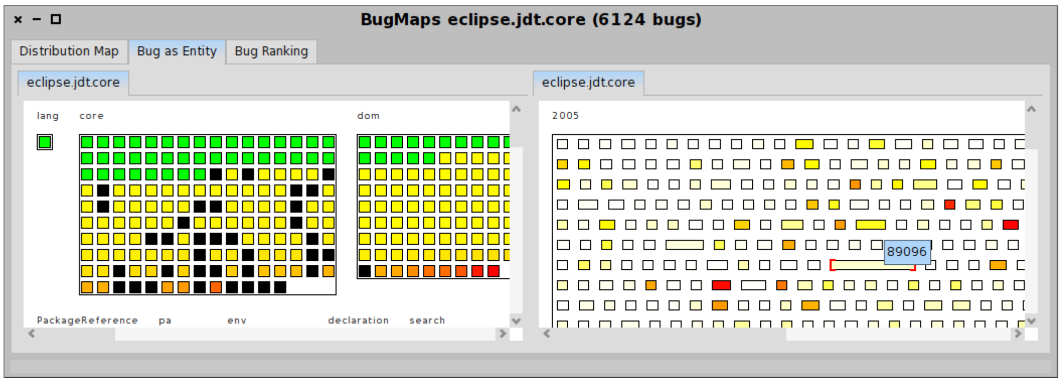
Figure 7.7: Granger results per class

**Bug as Entity:** As illustrated in Figure 7.8, this map represents bugs instead of classes. The color of a bug represents its lifetime, i.e., the number of days the bug remained open. Blue denotes a bug that was still open at the end of the considered time period. Moreover, white denotes a bug that was open for a short time. Similarly, yellow is used for a bug that was open up to three months, and red for a bug that was opened for more than three months. The width of a bug representation denotes its complexity, measured as the number of classes changed to fix the bug. Bugs are sorted according to the date they were created.

Figure 7.8(a) shows the bugs of the Equinox Framework created in 2010. We can observe that all bugs from 2010 were fixed (i.e., there are no bugs in blue), that only two bugs remained open for more than three months (bugs going to red), and that complex bugs (long width) are dispersed in time. Figure 7.8(b) shows the bugs of the Eclipse JDT Core created in 2005. Similar to the Equinox Framework, all bugs were fixed and few bugs remained open for more than three months. In addition, most bugs have low complexity (short width). However, in a detailed analysis, we can also



(a) Equinox Framework

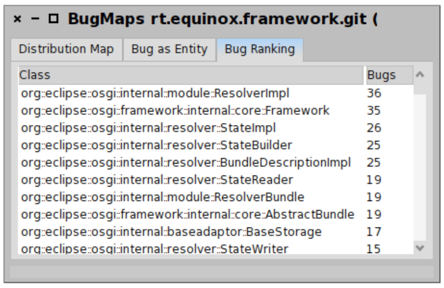


(b) Eclipse JDT Core

Figure 7.8: Bug as entity

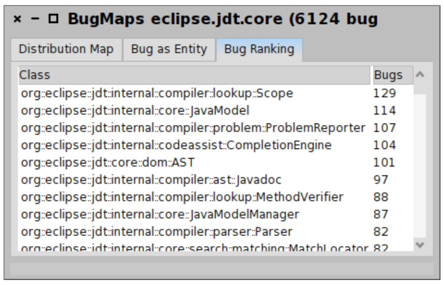
observe that the highlighted bug (ID 89096) is quite complex. More specifically, the developer team changed 75 classes in order to fix this particular bug, which is due to a performance problem in the resource bundle mechanism (a requirement scattered by the classes of the JDT Core).

**Bug Ranking:** Figure 7.9 shows examples of the Bug Ranking browser which is composed by one grid with two columns: class names (first column) and number of defects during the time frame considered in the analysis (second column). The figure shows the classes ranked by the defects for the Equinox Framework and Eclipse JDT Core systems during the period of three years. For the Equinox Framework, the two classes with more defects are `osgi.internal.module.ResolverImpl` and `osgi.framework.internal.core.Framework`. The `ResolverImpl` class—which has more than 2,000 lines of code—is an important class responsible for resolving the constraints of the bundles (JAR components) in a system that follows the OSGi standard [TV08]. The `Framework` class also has more than 2,000 lines of code and represents the core OSGi Framework class. For Eclipse JDT Core, the top-ten classes with more defects include classes such as `org.eclipse.jdt.core.dom.AST` and `org.eclipse.jdt.internal.core.JavaModel`. The `AST` class represents an abstract syntax tree node factory and the `JavaModel` class is one of the classes responsible for managing projects in the Eclipse workspace.



| Class | Bugs |
|---|---|
| org::eclipse::osgi::internal::module::ResolverImpl | 36 |
| org::eclipse::osgi::framework::internal::core::Framework | 35 |
| org::eclipse::osgi::internal::resolver::StateImpl | 26 |
| org::eclipse::osgi::internal::resolver::StateBuilder | 25 |
| org::eclipse::osgi::internal::resolver::BundleDescriptionImpl | 25 |
| org::eclipse::osgi::internal::resolver::StateReader | 19 |
| org::eclipse::osgi::internal::module::ResolverBundle | 19 |
| org::eclipse::osgi::framework::internal::core::AbstractBundle | 19 |
| org::eclipse::osgi::internal::baseadaptor::BaseStorage | 17 |
| org::eclipse::osgi::internal::resolver::StateWriter | 15 |

(a) Equinox Framework



| Class | Bugs |
|---|---|
| org::eclipse::jdt::internal::compiler::lookup::Scope | 129 |
| org::eclipse::jdt::internal::core::JavaModel | 114 |
| org::eclipse::jdt::internal::compiler::problem::ProblemReporter | 107 |
| org::eclipse::jdt::internal::codeassist::CompletionEngine | 104 |
| org::eclipse::jdt::core::dom::AST | 101 |
| org::eclipse::jdt::internal::compiler::ast::Javadoc | 97 |
| org::eclipse::jdt::internal::compiler::lookup::MethodVerifier | 88 |
| org::eclipse::jdt::internal::core::JavaModelManager | 87 |
| org::eclipse::jdt::internal::compiler::parser::Parser | 82 |
| org::eclipse::jdt::internal::core::search::matching::MatchLocator | 82 |

(b) Eclipse JDT Core

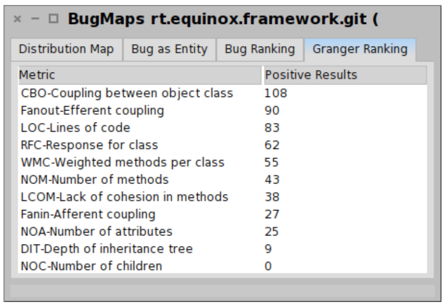
Figure 7.9: The top-10 defective classes

**Granger Ranking:** For each valid time series of source code metrics, Figure 7.10 shows the number of Granger tests that have returned a positive result. For example, the number of CBO time series with a Granger-causality with defects was 108 and 138 series for Equinox Framework and Eclipse JDT Core, respectively. As can be observed, the metrics with the highest number of positive Granger tests were CBO, FAN-OUT, LOC, RFC, and WMC for Equinox Framework; and RFC, LOC, WMC, and CBO for Eclipse JDT Core. On the other hand, the metrics with the lowest number of positive results were DIT and NOC for both systems.

Based on these results, we can conclude that metrics related to complexity (WMC), coupling (CBO, RFC, and FAN-OUT), and LOC tend to impact in the occurrence of defects in the Equinox Framework and Eclipse JDT Core systems, at least according to Granger. Conversely, metrics related to inheritance—such as DIT and NOC—tend to have a small influence in the occurrence of defects, at least in the considered systems.

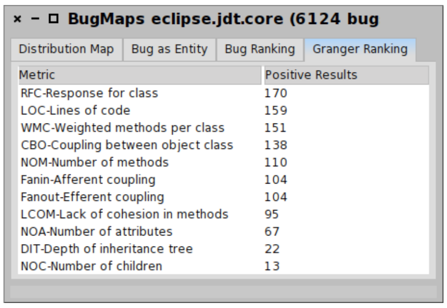
## 7.5 Related Tools

Churrasco is a web-based tool for collaborative software evolution analysis [DL10]. The tool automatically extracts information from a variety of software repositories, including versioning systems and bug management systems. The ultimate goal is to provide



| Metric | Positive Results |
|---|---|
| CBO-Coupling between object class | 108 |
| Fanout-Efferent coupling | 90 |
| LOC-Lines of code | 83 |
| RFC-Response for class | 62 |
| WMC-Weighted methods per class | 55 |
| NOM-Number of methods | 43 |
| LCOM-Lack of cohesion in methods | 38 |
| Fanin-Afferent coupling | 27 |
| NOA-Number of attributes | 25 |
| DIT-Depth of inheritance tree | 9 |
| NOC-Number of children | 0 |

(a) Equinox Framework



| Metric | Positive Results |
|---|---|
| RFC-Response for class | 170 |
| LOC-Lines of code | 159 |
| WMC-Weighted methods per class | 151 |
| CBO-Coupling between object class | 138 |
| NOM-Number of methods | 110 |
| Fanin-Afferent coupling | 104 |
| Fanout-Efferent coupling | 104 |
| LCOM-Lack of cohesion in methods | 95 |
| NOA-Number of attributes | 67 |
| DIT-Depth of inheritance tree | 22 |
| NOC-Number of children | 13 |

(b) Eclipse JDT Core

Figure 7.10: Number of time series with a positive result for Granger-causality

an extensible tool that can be used to reason about software evolution under different perspectives, including the behavior of bugs. In contrast, BugMaps-Granger provides information about source code properties (as measured by source code metrics) that caused bugs, at least according to Granger. Other visualization metaphors have also been provided for understanding the behavior of bugs, including system radiography (which provides a high-level visualization on the parts of the system more impacted by bugs) and bug watch (which relies on a watch metaphor to provide several information about a particular bug) [DLP07]. Hatari [SZZ05a] is a tool that provides views to browse through the most risky locations and to analyze the risk history of a particular component from a system at the level of lines of code. On the other hand, BugMaps-Granger works at the level of classes and packages. More recently, the tool in*Bug [SL13] was proposed to allow users navigating and inspecting the information stored in bug tracking systems, with the specific purpose to support the comprehension of bug reports.

## 7.6 Final Remarks

In this chapter, we described a tool that infers and provides visualizations about causality relations between source code metrics and bugs reported in bug tracking platforms. The BugMaps-Granger tool extracts time series of defects from such systems and allows the visualization of different bug measures, including the source code properties that Granger-caused bugs. The ultimate goal of BugMaps-Granger is to highlight changes in the source code that are more subjected to bugs, and the source code metrics that can be used to anticipate the occurrence of bugs in the changed classes. With this tool in hand, maintainers can perform at least two main actions for improving software quality: (a) refactoring the source code properties that Granger-caused bugs and (b) increasing unit tests in classes with more bugs.

The proposed tool and information on how to execute it is publicly available at:

http://aserg.labsoft.dcc.ufmg.br/bugmaps/

# Chapter 8

# Conclusion

In this chapter, we summary the results of this thesis (Section 8.1). Next, we review our contributions (Section 8.2). Finally, we present the further work (Section 8.3).

## 8.1 Summary

Defect prediction is a central challenge for software engineering research. The goal is to discover reliable predictors that can indicate in advance those components of a software system that are more likely to fail. For years, several works were conducted with the purpose of investigating relationships between software metrics and defects [BBM96, SK03, NB05a, NBZ06, MPS08, Has09]. This demonstrates that, although this issue is of great importance to the software development process, a solution widely used in real-world software development process is still to be reached.

In order to contribute to solving this problem, we designed, described, and evaluated a defect prediction approach centered on more robust evidences towards causality between source code metrics and the occurrence of defects. More specifically, we rely on the Granger Causality Test to evaluate whether past variations in source code metrics values can be used to forecast changes in time series of defects. Our approach triggers alarms when changes made to the source code of a target system have a high chance of producing defects.

In the feasibility study reported in Chapter 5, we concluded that, even by reducing our sample to 18% of the classes after applying the preconditions and the Granger Test, it was possible to cover 73% of the defects reported for the systems considered in our dataset. In Chapter 6, we evaluated our approach in several life stages of four Java-based systems. We reached an average precision greater than 50% in three out of the four systems we evaluated.

On the other hand, as expected, we were not able to trigger alarms for all defects using times series of source code metrics as predictors. On average, we achieved recall rates ranging from 13% (Equinox Framework) to 31% (Lucene). We argue that the main reason is the fact that there is a large spectrum of bugs that can be reported for any system. Probably, some types of bugs are less impacted by variations in the values of the source code metrics. Finally, we showed that our models outperform models that trigger alarms without considering Granger-causality or that are based on linear regression techniques.

## 8.2 Contributions

This research makes the following contributions:

- An evaluation of the effectiveness of the warnings issued by a bug finding tool to predict software defects (Section 2.3). We concluded that there is no static correspondence between field defects and warnings generated by FindBugs.
- An dataset for defect prediction based on historical data that includes time series of source code metrics and defects for four real-world Java systems (Chapter 4).
- An methodology to systematically mine for Granger-causality relationships between the values of source code metrics and defects (Chapter 5). After using Granger to mine for such relationships, we were able to associate to the historical values of metrics the causes of 73% of the defects reported for the systems considered in the aforementioned dataset.
- A defect prediction model centered on causal relations between time series of source code metrics and software defects (Section 6.3). This model triggers alarms when changes made to a target system have a high chance of producing defects.
- An evaluation of the proposed model in several life stages of the systems included in the aforementioned dataset (Section 6.4). Our model reached an average precision greater than 50% in three out of the four systems we evaluated. Furthermore, by comparing our approach with baselines that are not based on causality tests, the proposed model achieved a better precision.
- A prototype tool called BugMaps-Granger publicly available for the visual exploration and analysis of bugs and for the detection of causal relations between source code metrics and bugs (Chapter 7).

## 8.3 Further Work

This work must be complemented by the following future work:

- The design and implementation of a tool that alerts maintainers about future occurrence of defects. We suggest to implement this tool as a plug-in for version control platforms, like SVN and Git. Basically, this tool should trigger alarms whenever risky changes are committed to the version control platform. Based on such alarms, the maintainer can for example perform software quality assurance activities (e.g., testing or code inspections) before executing the commit.

- An extension of the proposed defect prediction model to consider other internal software quality metrics, including violations in the static architecture of software systems, as revealed by the DCL language [TV09] or the ArchLint tool [MVA+13]. Is is also interesting to investigate the relations between defects and code smells. In this case, we intend to start by investigating the relations between defects and methods located in inappropriate classes (i.e., feature envy instances), as revealed by the JMove recommendation system [STMV13].

- An extension of the proposed defect prediction model to handle the cases where changes in a class cause defects in other classes of the system.

- The development of a qualitative analysis on why some defects can be predicted and others not, which certainly requires a direct participation of expert developers on the target systems.

# Bibliography

[ACSV10] João Eduardo Araujo, Cesar Couto, Silvio Souza, and Marco Tulio Valente. Um estudo sobre a correlação entre defeitos de campo e warnings reportados por uma ferramenta de análise estática. In *Simpósio Brasileiro de Qualidade de Software (SBQS)*, pages 9–23, 2010.

[ASV11] Joao Eduardo Araujo, Silvio Souza, and Marco Tulio Valente. Study on the relevance of the warnings reported by Java bug finding tools. *IET Software*, 5(4):366–374, 2011.

[BBM96] Victor R. Basili, Lionel C. Briand, and Walcelio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[BDW99] Lionel C. Briand, John W. Daly, and Jurgen Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[BWD$^+$00] Lionel C. Briand, Jürgen Wüst, John W. Daly, , and Victor D. Porter. Exploring the relationship between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3):245–273, 2000.

[BWIL99] Lionel C. Briand, Jürgen Wüst, Stefan V. Ikonomovski, and Hakim Lounis. Investigating quality factors in object-oriented designs: an industrial case study. In *International Conference on Software Engineering*, pages 345–354, 1999.

[CASV13] Cesar Couto, Joao Eduardo Araujo, Christofer Silva, and Marco Tulio Valente. Static correspondence and correlation between field defects and warnings reported by a bug finding tool. *Software Quality Journal*, 21(2):241–257, 2013.

[CCPC10] Gerardo Canfora, Michele Ceccarelli, Massimiliano Di Penta, and Luigi Cerulo. Using multivariate time series and association rules to detect logical change coupling: an empirical study. In *International Conference on Software Maintenance (ICSM)*, pages 1–10, 2010.

[CK91] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *International Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 197–211, 1991.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[CMGV13] Cesar Couto, Cristiano Maffort, Rogel Garcia, and Marco Tulio Valente. Comets: A dataset for empirical research on software evolution using source code metrics and time series analysis. *Software Engineering Notes*, pages 1–3, 2013.

[Cop05] Tom Copeland. *PMD Applied*. Centennial Books, 2005.

[Cor89] T. A. Corbi. Program understanding: challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.

[CPV$^+$13] Cesar Couto, Pedro Pires, Marco Tulio Valente, Roberto S. Bigonha, Andre Hora, and Nicolas Anquetil. Bugmaps-granger: A tool for causality analysis between source code metrics and bugs. In *Brazilian Conference on Software: Theory and Practice (Tools Session)*, 2013.

[CS00] Michelle Cartwright and Martin Shepperd. An empirical investigation of an object-oriented software system. *IEEE Transactions on Software Engineering*, 26(8):786–796, 2000.

[CSV$^+$12] Cesar Couto, Christofer Silva, Marco Tulio Valente, Roberto Bigonha, and Nicolas Anquetil. Uncovering causal relationships between software metrics and bugs. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pages 223–232, 2012.

[CVB11] Cesar Couto, Marco Tulio Valente, and Roberto Bigonha. Avaliação de causalidade entre métricas de qualidade interna e defeitos. In *Simpósio Brasileiro de Qualidade de Software (SBQS)*, pages 1–15, 2011.

[DAB$^+$11] Stéphane Ducasse, Nicolas Anquetil, Muhammad Usman Bhatti, Andre Hora, Jannik Laval, and Tudor Girba. MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. Technical report, RMOD - INRIA Lille - Nord Europe , Software Composition Group - SCG, 2011.

[DGK06] Stephane Ducasse, Tudor Girba, and Adrian Kuhn. Distribution Map. In *International Conference on Software Maintenance (ICSM)*, pages 203–212, 2006.

[DL07] Marco D'Ambros and Michele Lanza. Bugcrawler: Visualizing evolving software systems. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pages 333–334, 2007.

[DL10] Marco D'Ambros and Michele Lanza. Distributed and collaborative software evolution analysis with churrasco. *Science of Computer Programming*, 75(4):276–287, 2010.

[DLP07] Marco D'Ambros, Michele Lanza, and Martin Pinzger. A bug's life: Visualizing a bug database. In *International Workshop on Visualizing Software for Analysis and Understanding (VISSOFT)*, pages 113–120, 2007.

[DLR10] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Working Conference on Mining Software Repositories (MSR)*, pages 31–41, 2010.

[DLR12] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Journal of Empirical Software Engineering*, 17(4-5):531–577, 2012.

[EMM01] Khaled El Emam, Walcelio Melo, and Javam C. Machado. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56(1):63–75, 2001.

[FP97] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* PWS Publishing Company, 1997.

[Ful94] Wayne A. Fuller. *Introduction to Statistical Time Series.* John Wiley & Sons, 1994.

[GD06] Tudor Girba and Stéphane Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 18:207–236, 2006.

[GDPG12] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C. Gall. Method-level bug prediction. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 171–180, 2012.

[GFS05] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transaction on Software Engineering*, 31(10):897–910, 2005.

[GKMS00] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[Gra69] Clive Granger. Investigating causal relations by econometric models and cross-spectral methods. *Econometrica*, 37(3):424–438, 1969.

[Gra81] Clive Granger. Some properties of time series data and their use in econometric model specification. *Journal of Econometrics*, 16(6):121–130, 1981.

[Has09] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *International Conference on Software Engineering (ICSE)*, pages 78–88, 2009.

[HBB$^+$12] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[HCA$^+$12] Andre Hora, Cesar Couto, Nicolas Anquetil, Stephane Ducasse, Muhammad Bhatti, Marco Tulio Valente, and Julio Martins. Bugmaps: A tool for the visual exploration and analysis of bugs. In *European Conference on Software Maintenance and Reengineering (CSMR Tool Demonstration)*, 2012.

[HL00] David W. Hosmer and Stanley Lemeshow. *Applied Logistic Regression.* Wiley, 2000.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[HPH$^+$09] Tilman Holschuh, Markus Pauser, Kim Herzig, Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects in SAP Java code:

An experience report. In *International Conference on Software Engineering (ICSE)*, pages 172–181, 2009.

[Hum95] Watts S. Humphrey. *A Discipline for Software Engineering.* Addison-Wesley, 1995.

[Kan02] Stephen H. Kan. *Metrics and Models in Software Quality Engineering.* Addison-Wesley, 2002.

[KSA$^+$13] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2013.

[Lan01] Michele Lanza. The evolution matrix: recovering software evolution using software visualization techniques. In *International Workshop on Principles of Software Evolution (IWPSE)*, pages 37–42, 2001.

[LB85] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change.* Academic Press Professional, 1985.

[Leh80] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[LMD05] Michele Lanza, Radu Marinescu, and Stephane Ducasse. *Object-Oriented Metrics in Practice.* Springer-Verlag, 2005.

[LS80] Bennet P Lientz and E Burton Swanson. *Software Maintenance Management.* Addison-Wesley, 1980.

[MC07] Nitai D. Mukhopadhyay and Snigdhansu Chatterjee. Causality and pathway search in microarray time series experiment. *Bioinformatics*, 23(4):442–449, 2007.

[Mey00] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2000.

[MPS08] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *International Conference on Software Engineering (ICSE)*, pages 181–190, 2008.

[MVA$^+$13] Cristiano Maffort, Marco Tulio Valente, Nicolas Anquetil, Andre Hora, and Mariza Bigonha. Heuristics for discovering architectural violations. In *Working Conference on Reverse Engineering (WCRE)*, pages 222–231, 2013.

[NB05a] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *International Conference on Software Engineering (ICSE)*, pages 580–586, 2005.

[NB05b] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *International Conference on Software Engineering (ICSE)*, pages 284–292, 2005.

[NBZ06] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *International Conference on Software Engineering (ICSE)*, pages 452–461, 2006.

[NDG05a] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gĭrba. The story of Moose: an agile reengineering environment. In *European Software Engineering Conference (ESEC)*, pages 1–10, 2005.

[NDG05b] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gĭrba. The story of moose: an agile reengineering environment. *Software Engineering Notes*, 30:1–10, 2005.

[NP90] J. T. Nosek and P. Palvia. Software maintenance management: changes in the last decade. *Journal of Software Maintenance*, 2(3):157–174, 1990.

[Pea85] Judea Pearl. Bayesian networks: A model of self-activated memory for evidential reasoning. In *Cognitive Science Society*, pages 329–334, 1985.

[Pea00] Judea Pearl. *Causality: models, reasoning, and inference*. Cambridge University Press, 2000.

[PPV97] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. A primer on empirical studies (tutorial). In *Tutorial presented at International Conference on Software Engineering (ICSE)*, pages 657–658, 1997.

[Pre10] Roger S Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2010.

[RCM$^+$12] Henrique Rocha, Cesar Couto, Cristiano Maffort, Rogel Garcia, Clarisse Simoes, Leonardo Passos, and MarcoTulio Valente. Mining the impact of evolution categories on object-oriented metrics. *Software Quality Journal*, pages 1–21, 2012.

[Sch89] G. William Schwert. Tests for unit roots: A monte carlo investigation. *Journal of Business & Economic Statistics*, 7(2):147–159, 1989.

[SK03] Ramanath Subramanyam and M. S. Krishnan. Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Transaction on Software Engineering*, 29(4):297–310, 2003.

[SL13] Tommaso Dal Sassc and Michele Lanza. A closer look at bugs. In *Working Conference on Software Visualization (VISSOFT)*, pages 1–4, 2013.

[Som09] Ian Sommerville. *Software Engineering*. Addison-Wesley, 2009.

[Son13] SonarSource. Sonar platform, 2013.

[SPUP02] Andrew I. Schein, Alexandrin Popescul, Lyle H. Ungar, and David M. Pennock. Methods and metrics for cold-start recommendations. In *International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 253–260, 2002.

[Sta90] IEEE Standard. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12, 1990.

[STMV13] Vitor Sales, Ricardo Terra, Luis Fernando Miranda, and Marco Tulio Valente. Recommending move method refactorings using dependency sets. In *Working Conference on Reverse Engineering (WCRE)*, pages 232–241, 2013.

[SZZ05a] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. Hatari: Raising risk awareness. In *European Software Engineering Conference (ESEC)*, pages 107–110, 2005.

[SZZ05b] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Working Conference on Mining Software Repositories (MSR)*, pages 1–5, 2005.

[SZZ06] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *International Symposium on Empirical Software Engineering (ISESE)*, pages 18–27, 2006.

[TAD$^+$10] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *Asia Pacific Software Engineering Conference (APSEC)*, pages 336–345, 2010.

[TMVB13] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S. Bigonha. Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *Software Engineering Notes*, 38(5):1–4, 2013.

[Tri06] Mario Triola. *Elementary Statistics.* Addison-Wesley, 2006.

[TV08] Andre Tavares and Marco Tulio Valente. A gentle introduction to OSGi. *ACM SIGSOFT Software Engineering Notes*, 33(5):1–5, 2008.

[TV09] Ricardo Terra and Marco Tulio Valente. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 32(12):1073–1094, 2009.

[Vis10] Joost Visser. Research shows strong impact of software quality on cost. *InSIGht*, September 2010.

[VLJ10] Rajesh Vasa, Markus Lumpe, and Allan Jones. Helix-software evolution data set, 2010. http://www.ict.swin.edu.au/research/pro jects/helix.

[WAWS08] Stefan Wagner, Michael Aichner, Johann Wimmer, and Markus Schwalb. An evaluation of two bug pattern tools for Java. In *International Conference on Software Testing, Verification, and Validation (ICST)*, pages 248–257, 2008.

[Wet09] Richard Wettel. Visual exploration of large-scale evolving software. In *International Conference on Software Engineering (ICSE)*, pages 391–394, 2009.

[ZNZ08] Thomas Zimmermann, Nachiappan Nagappan, and Andreas Zeller. *Predicting Bugs from History*, chapter 4, pages 69–88. Springer, 2008.

[ZPZ07] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for Eclipse. In *International Workshop on Predictor Models in Software Engineering*, page 9, 2007.

[ZWN$^+$06] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, John P. Hudepohl, and Mladen A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4), 2006.