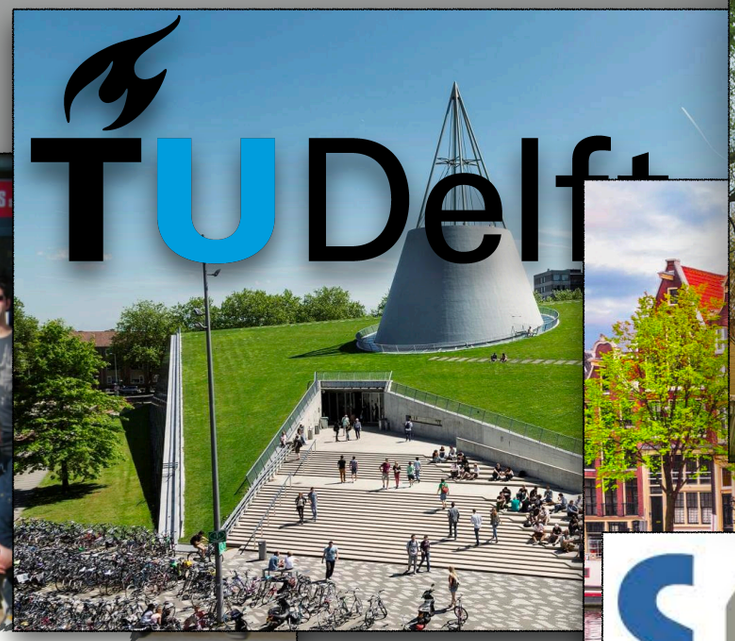
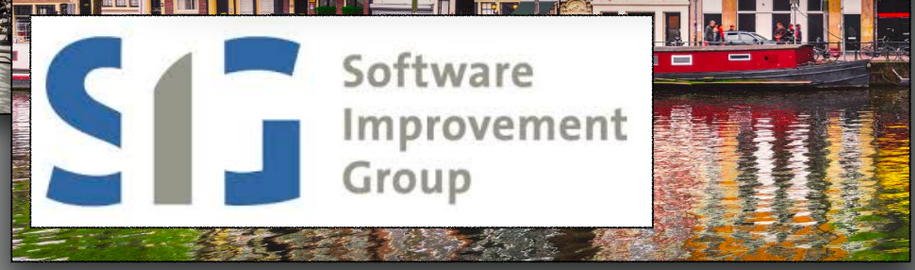
Software Defect Prediction

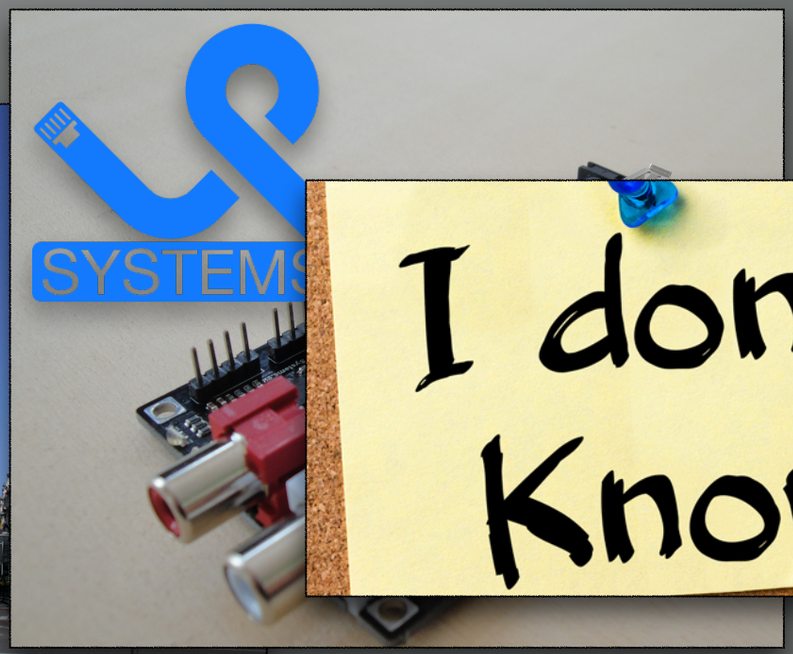
Luca Pascarella

Who is Luca?





2008 2010 2012 2015 2016 2017 2018 2019 2020

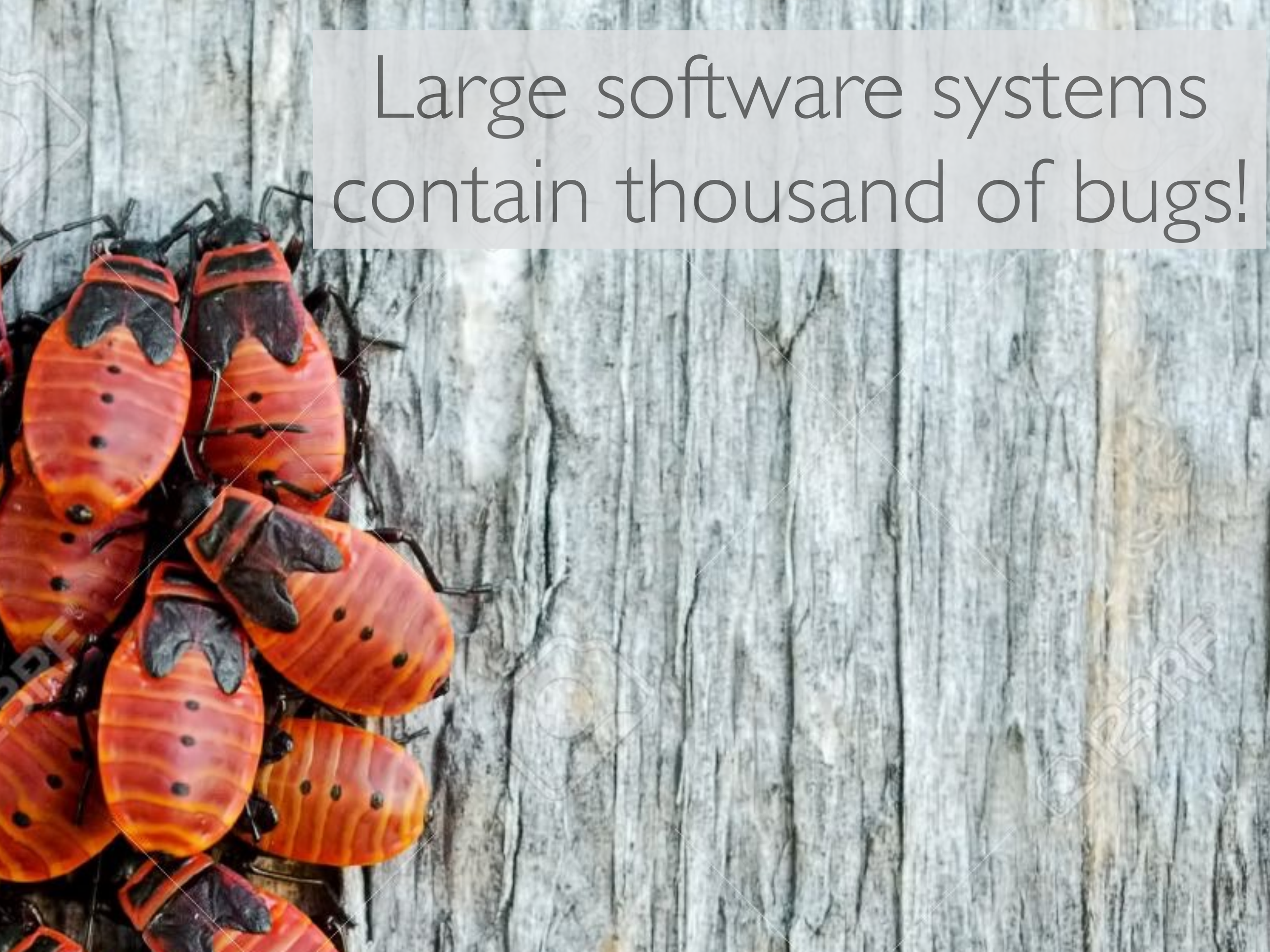


Software Defect Prediction

An Overview

Luca Pascarella and Fabio Palomba


Developers usually
introduce bugs



Large software systems contain thousand of bugs!

But we have limited resources

Testing an entire system is not feasible

[Menzies et al. - 2007]

Inspecting source code is costly as well

[Rahman - 2011]

Thus, we need to predict which components are more prone to be subject of bugs!

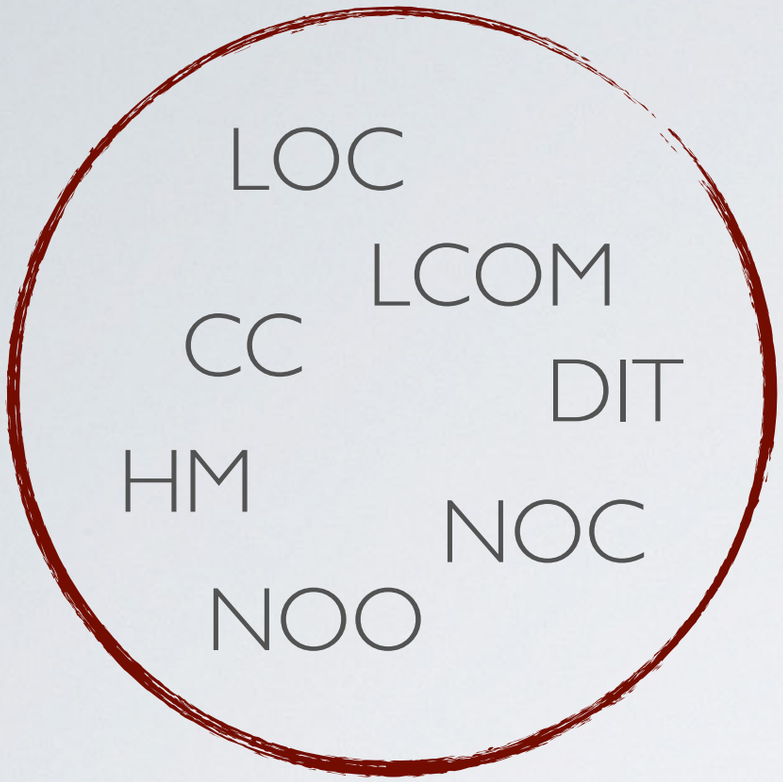
The higher the **complexity**
of a code component

ASSUMPTION

The higher the **complexity** of a code component

The higher the **bug-proneness** of a code component

ASSUMPTION



Predictors

HOW?

LOC

LCOM

CC

DIT

HM

NOC

NOO

Predictors

Machine Learner

HOW?

LOC

LCOM

CC

DIT

HM

NOC

NOO

Predictors

Machine Learner

Buggy / Non-buggy

HOW?

LOC

LCOM

CC

DIT

HM

NOC

NOO

Predictors

Machine Learner

Buggy / Non-buggy

Ranking

HOW?

The choice of the predictors to use for the prediction

Research provides different alternatives

- Structural metrics
- Complexity metrics
- Process metrics
- Developer-based metrics
- Textual metrics

Research provides different alternatives

- Structural metrics
- Complexity metrics
- Process metrics
- Developer-based metrics
- Textual metrics

The Akiyama's equation

Application of linear regression analysis to measure the number of bugs

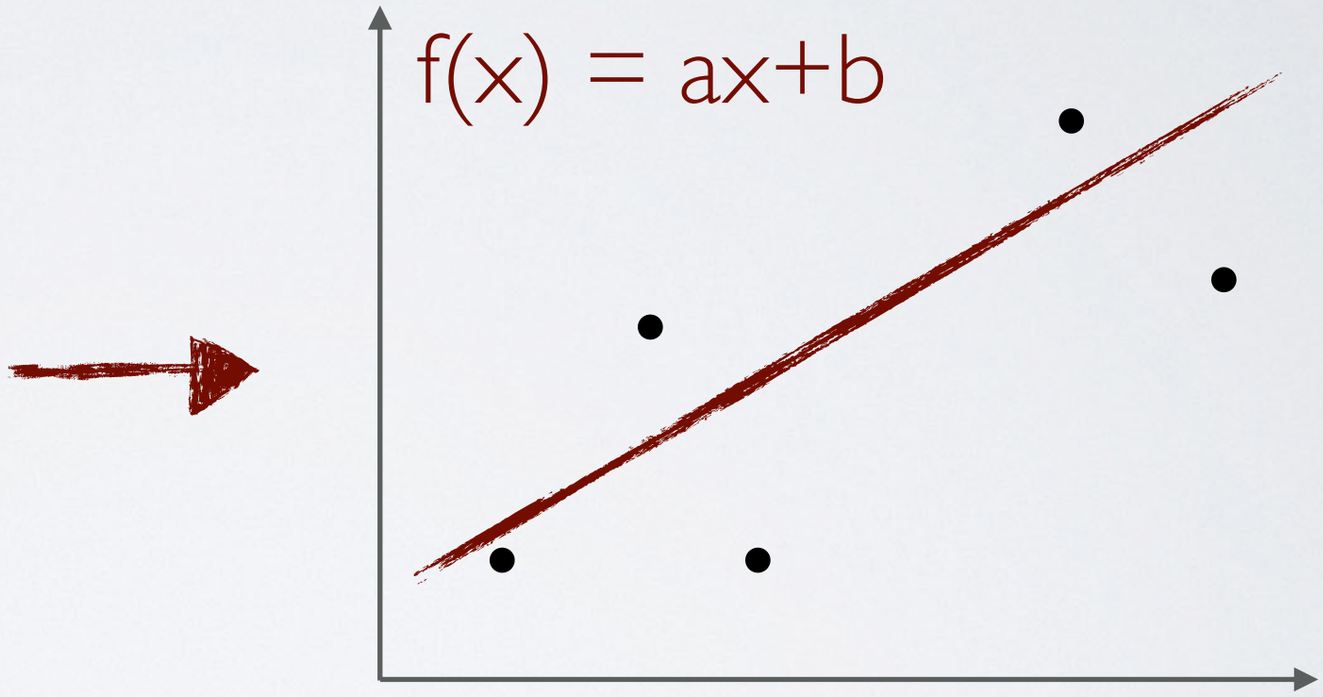
Fitting models

1970s

The Akiyama's equation

Application of linear regression analysis to measure the number of bugs

LOC	# Bug
100	2
12.093	1
26.761	4
27.565	3



Fitting models

1970s

The Akiyama's equation

bugs = 4.86 + 0.018 * LOC

Fitting models

1970s

CK Metrics

A Metrics Suite for Object Oriented Design

Shyam R. Chidamber and Chris F. Kemerer

Abstract— **Given the central role that software development plays in the delivery and application of information technology, managers are increasingly focusing on process improvement in the software development area. This demand has spurred the provision of a number of new and/or improved approaches to software development, with perhaps the most prominent being object-orientation (OO). In addition, the focus on process improvement has increased the demand for software measures, or metrics with which to manage the process. The need for such metrics is particularly acute when an organization is adopting a new technology for which established practices have yet to be developed. This research addresses these needs through the development and implementation of a new suite of metrics for OO design. Metrics developed in previous research, while contributing to the field's understanding of software development processes, have generally been subject to serious criticisms, including the lack of a theoretical base. Following Wand and Weber, the theoretical base chosen for the metrics was the ontology of Bunge. Six design metrics are developed, and then analytically evaluated against Weyuker's proposed set of measurement principles. An automated data collection tool was then developed and implemented to collect an empirical sample of these metrics at two field sites in order to demonstrate their feasibility and suggest ways in which managers may use these metrics for process improvement.**

Index Terms— **CR categories and subject descriptors: D.2.8 [software engineering]: metrics; D.2.9 [software engineering]: management; F.2.3 [analysis of algorithms and problem complexity]: tradeoffs among complexity measures; K.6.3 [management of computing and information systems]: software management. General terms: Class, complexity, design, management, measurement, metrics, object orientation, performance.**

I. Introduction

IT has been widely recognized that an important component of process improvement is the ability to measure the process. Given the central role that software development plays in the delivery and application of information technology, managers are increasingly focusing on process improvement in the software development area. This emphasis has had two effects. The first is that this demand has spurred the provision of a number of new and/or improved approaches to software development, with perhaps the most prominent being object-orientation (OO). Second, the focus on process improvement has increased the demand for software measures, or metrics with which to manage the process. The need for such metrics is particularly acute when an organization is adopting a new technology for which established practices have yet to be developed.

This research addresses these needs through the development and implementation of a new suite of metrics for OO design. Previous research on software metrics, while contributing to the field's understanding of software development processes, have generally been subject to one or more types of criticisms. These include: lacking a theoretical basis [41], lacking in desirable measurement properties [47], being insufficiently generalized or too implementation technology dependent [45], and being too labor-intensive to collect [22].

Following Wand and Weber, the theoretical base chosen for the OO design metrics was the ontology of Bunge [5], [6], [43]. Six design metrics were developed, and analytically evaluated against a previously proposed set of measurement principles. An automated data collection tool was then developed and implemented to collect an empirical sample of these metrics at two field sites in order to demonstrate their feasibility and to suggest ways in which managers may use these metrics for process improvement.

The key contributions of this paper are the development and empirical validation of a set of theoretically-grounded metrics of OO design. The rest of this paper is organized as follows. The next section presents a brief summary of the research problem, followed by a section describing the theory underlying the approach taken. Then Weyuker's list of software metric evaluation criteria is presented, along with a brief description of the empirical data collection sites. The Results section presents the metrics, their analytical evaluation, the empirical data and a managerial interpretation of the data for each metric. Some concluding remarks are presented in the final section.

II. Research Problem

There are two general types of criticisms that can be applied to current software metrics. The first category are those theoretical criticisms that are leveled at conventional software metrics as they are applied to traditional, non-OO software design and development. Kearney, et al. criticized software complexity metrics as being without solid theoretical bases and lacking appropriate properties [21]. Vessey and Weber also commented on the general lack of theoretical rigor in the structured programming literature [41]. Both Prather and Weyuker proposed that traditional software complexity metrics do not possess appropriate mathematical properties, and consequently fail to display what might be termed normal predictable behavior [34], [47]. This suggests that software

Manuscript received February 17, 1993; revised January, 1994. recommended by S. Zweben. This research was supported in part by the M.I.T. Center for Information Systems Research (CISR), and the cooperation of two industrial organizations who supplied the data.

The authors are with the Massachusetts Institute of Technology, E53-315, 30 Wadsworth Street, Cambridge, MA 02139 USA; e-mail: shyam@athena.mit.edu or ckemerer@sloan.mit.edu.

IEEE Log Number 9401707.

Definition of a set of software metrics for OO programming

1994

CK Metrics

A Metrics Suite for Object Oriented Design

Shyam R. Chidamber and Chris F. Kemerer

Abstract— **Given the central role that software development plays in the delivery and application of information technology, managers are increasingly focusing on process improvement in the software development area. This demand has spurred the provision of a number of new and/or improved approaches to software development, with perhaps the most prominent being object-orientation (OO). In addition, the focus on process improvement has increased the demand for software measures, or metrics with which to manage the process. The need for such metrics is particularly acute when an organization is adopting a new technology for which established practices have yet to be developed. This research addresses these needs through the development and implementation of a new suite of metrics for OO design. Metrics developed in previous research, while contributing to the field's understanding of software development processes, have generally been subject to serious criticisms, including the lack of a theoretical base. Following Wand and Weber, the theoretical base chosen for the metrics was the ontology of Bunge. Six design metrics are developed, and then analytically evaluated against Weyuker's proposed set of measurement principles. An automated data collection tool was then developed and implemented to collect an empirical sample of these metrics at two field sites in order to demonstrate their feasibility and suggest ways in which managers may use these metrics for process improvement.**

Index Terms— **CR categories and subject descriptors: D.2.8 [software engineering]: metrics; D.2.9 [software engineering]: management; F.2.3 [analysis of algorithms and problem complexity]: tradeoffs among complexity measures; K.6.3 [management of computing and information systems]: software management. General terms: Class, complexity, design, management, measurement, metrics, object orientation, performance.**

I. Introduction

IT has been widely recognized that an important component of process improvement is the ability to measure the process. Given the central role that software development plays in the delivery and application of information technology, managers are increasingly focusing on process improvement in the software development area. This emphasis has had two effects. The first is that this demand has spurred the provision of a number of new and/or improved approaches to software development, with perhaps the most prominent being object-orientation (OO). Second, the focus on process improvement has increased the demand for software measures, or metrics with which to manage the process. The need for such metrics is particularly acute when an organization is adopting a new technology for which established practices have yet to be developed.

This research addresses these needs through the development and implementation of a new suite of metrics for OO design. Previous research on software metrics, while contributing to the field's understanding of software development processes, have generally been subject to one or more types of criticisms. These include: lacking a theoretical basis [41], lacking in desirable measurement properties [47], being insufficiently generalized or too implementation technology dependent [45], and being too labor-intensive to collect [22].

Following Wand and Weber, the theoretical base chosen for the OO design metrics was the ontology of Bunge [5], [6], [43]. Six design metrics were developed, and analytically evaluated against a previously proposed set of measurement principles. An automated data collection tool was then developed and implemented to collect an empirical sample of these metrics at two field sites in order to demonstrate their feasibility and to suggest ways in which managers may use these metrics for process improvement.

The key contributions of this paper are the development and empirical validation of a set of theoretically-grounded metrics of OO design. The rest of this paper is organized as follows. The next section presents a brief summary of the research problem, followed by a section describing the theory underlying the approach taken. Then Weyuker's list of software metric evaluation criteria is presented, along with a brief description of the empirical data collection sites. The Results section presents the metrics, their analytical evaluation, the empirical data and a managerial interpretation of the data for each metric. Some concluding remarks are presented in the final section.

II. Research Problem

There are two general types of criticisms that can be applied to current software metrics. The first category are those theoretical criticisms that are leveled at conventional software metrics as they are applied to traditional, non-OO software design and development. Kearney, et al. criticized software complexity metrics as being without solid theoretical bases and lacking appropriate properties [21]. Vessey and Weber also commented on the general lack of theoretical rigor in the structured programming literature [41]. Both Prather and Weyuker proposed that traditional software complexity metrics do not possess appropriate mathematical properties, and consequently fail to display what might be termed normal predictable behavior [34], [47]. This suggests that software

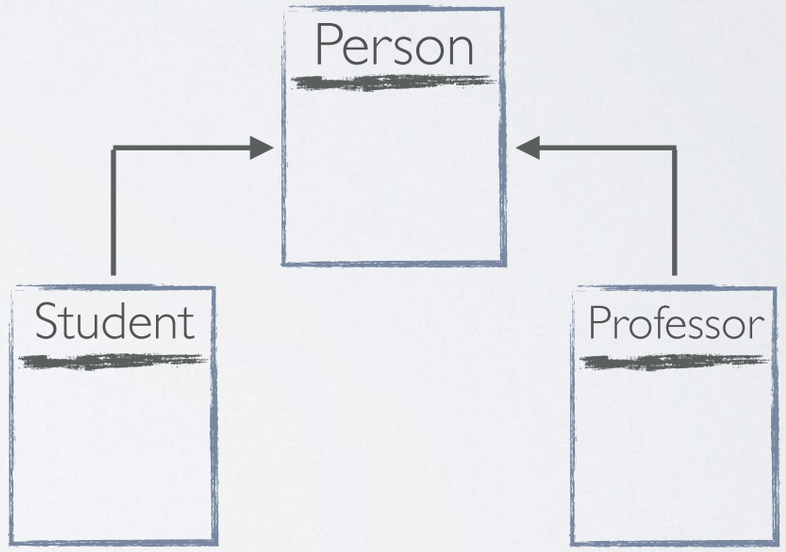
Manuscript received February 17, 1993; revised January, 1994. recommended by S. Zweben. This research was supported in part by the M.I.T. Center for Information Systems Research (CISR), and the cooperation of two industrial organizations who supplied the data.

The authors are with the Massachusetts Institute of Technology, E53-315, 30 Wadsworth Street, Cambridge, MA 02139 USA; e-mail: shyam@athena.mit.edu or ckemerer@sloan.mit.edu.

IEEE Log Number 9401707.

Definition of a set of software metrics for OO programming

Number Of Children (NOC)



1994

CK Metrics

A Metrics Suite for Object Oriented Design

Shyam R. Chidamber and Chris F. Kemerer

Abstract— **Given the central role that software development plays in the delivery and application of information technology, managers are increasingly focusing on process improvement in the software development area. This demand has spurred the provision of a number of new and/or improved approaches to software development, with perhaps the most prominent being object-orientation (OO). In addition, the focus on process improvement has increased the demand for software measures, or metrics with which to manage the process. The need for such metrics is particularly acute when an organization is adopting a new technology for which established practices have yet to be developed. This research addresses these needs through the development and implementation of a new suite of metrics for OO design. Metrics developed in previous research, while contributing to the field's understanding of software development processes, have generally been subject to serious criticisms, including the lack of a theoretical base. Following Wand and Weber, the theoretical base chosen for the metrics was the ontology of Bunge. Six design metrics are developed, and then analytically evaluated against Weyuker's proposed set of measurement principles. An automated data collection tool was then developed and implemented to collect an empirical sample of these metrics at two field sites in order to demonstrate their feasibility and suggest ways in which managers may use these metrics for process improvement.**

Index Terms— **CR categories and subject descriptors: D.2.8 [software engineering]: metrics; D.2.9 [software engineering]: management; F.2.3 [analysis of algorithms and problem complexity]: tradeoffs among complexity measures; K.6.3 [management of computing and information systems]: software management. General terms: Class, complexity, design, management, measurement, metrics, object orientation, performance.**

I. Introduction

IT has been widely recognized that an important component of process improvement is the ability to measure the process. Given the central role that software development plays in the delivery and application of information technology, managers are increasingly focusing on process improvement in the software development area. This emphasis has had two effects. The first is that this demand has spurred the provision of a number of new and/or improved approaches to software development, with perhaps the most prominent being object-orientation (OO). Second, the focus on process improvement has increased the demand for software measures, or metrics with which to manage the process. The need for such metrics is particularly acute when an organization is adopting a new technology for which established practices have yet to be developed.

This research addresses these needs through the development and implementation of a new suite of metrics for OO design. Previous research on software metrics, while contributing to the field's understanding of software development processes, have generally been subject to one or more types of criticisms. These include: lacking a theoretical basis [41], lacking in desirable measurement properties [47], being insufficiently generalized or too implementation technology dependent [45], and being too labor-intensive to collect [22].

Following Wand and Weber, the theoretical base chosen for the OO design metrics was the ontology of Bunge [5], [6], [43]. Six design metrics were developed, and analytically evaluated against a previously proposed set of measurement principles. An automated data collection tool was then developed and implemented to collect an empirical sample of these metrics at two field sites in order to demonstrate their feasibility and to suggest ways in which managers may use these metrics for process improvement.

The key contributions of this paper are the development and empirical validation of a set of theoretically-grounded metrics of OO design. The rest of this paper is organized as follows. The next section presents a brief summary of the research problem, followed by a section describing the theory underlying the approach taken. Then Weyuker's list of software metric evaluation criteria is presented, along with a brief description of the empirical data collection sites. The Results section presents the metrics, their analytical evaluation, the empirical data and a managerial interpretation of the data for each metric. Some concluding remarks are presented in the final section.

II. Research Problem

There are two general types of criticisms that can be applied to current software metrics. The first category are those theoretical criticisms that are leveled at conventional software metrics as they are applied to traditional, non-OO software design and development. Kearney, et al. criticized software complexity metrics as being without solid theoretical bases and lacking appropriate properties [21]. Vessey and Weber also commented on the general lack of theoretical rigor in the structured programming literature [41]. Both Prather and Weyuker proposed that traditional software complexity metrics do not possess appropriate mathematical properties, and consequently fail to display what might be termed normal predictable behavior [34], [47]. This suggests that software

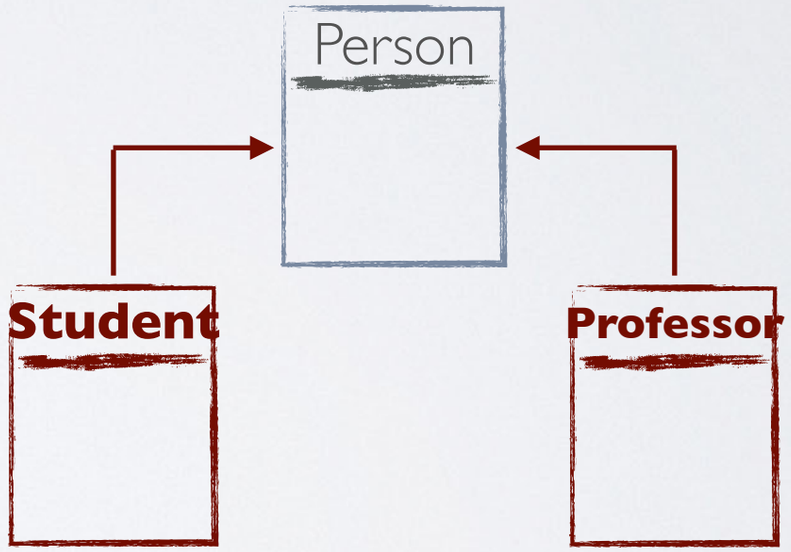
Manuscript received February 17, 1993; revised January, 1994. recommended by S. Zweben. This research was supported in part by the M.I.T. Center for Information Systems Research (CISR), and the cooperation of two industrial organizations who supplied the data.

The authors are with the Massachusetts Institute of Technology, E53-315, 30 Wadsworth Street, Cambridge, MA 02139 USA; e-mail: shyam@athena.mit.edu or ckemerer@sloan.mit.edu.

IEEE Log Number 9401707.

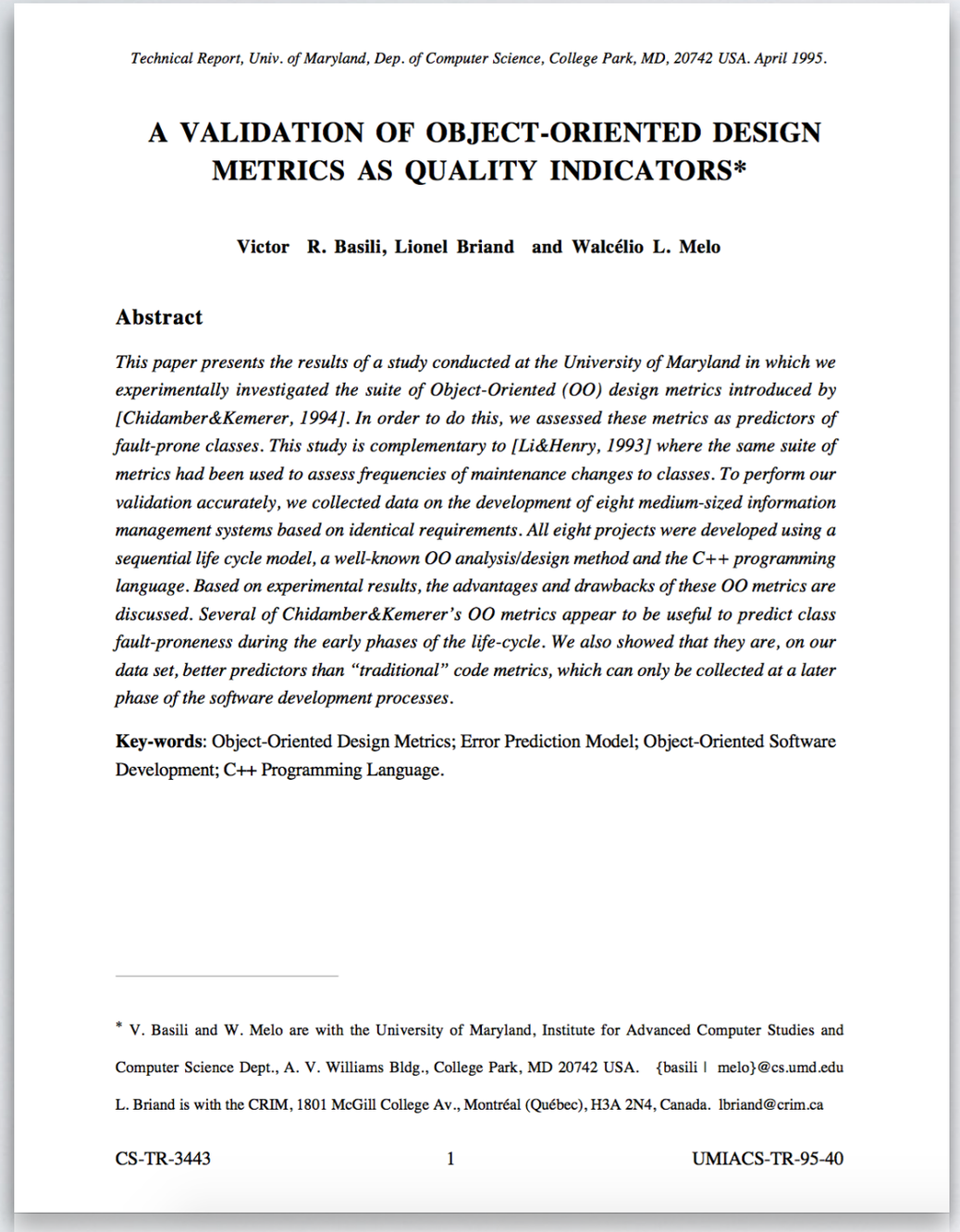
Definition of a set of software metrics for OO programming

Number Of Children (NOC)



1994

CK Metrics



Technical Report, Univ. of Maryland, Dep. of Computer Science, College Park, MD, 20742 USA. April 1995.

A VALIDATION OF OBJECT-ORIENTED DESIGN METRICS AS QUALITY INDICATORS*

Victor R. Basili, Lionel Briand and Walcélio L. Melo

Abstract

This paper presents the results of a study conducted at the University of Maryland in which we experimentally investigated the suite of Object-Oriented (OO) design metrics introduced by [Chidamber&Kemerer, 1994]. In order to do this, we assessed these metrics as predictors of fault-prone classes. This study is complementary to [Li&Henry, 1993] where the same suite of metrics had been used to assess frequencies of maintenance changes to classes. To perform our validation accurately, we collected data on the development of eight medium-sized information management systems based on identical requirements. All eight projects were developed using a sequential life cycle model, a well-known OO analysis/design method and the C++ programming language. Based on experimental results, the advantages and drawbacks of these OO metrics are discussed. Several of Chidamber&Kemerer's OO metrics appear to be useful to predict class fault-proneness during the early phases of the life-cycle. We also showed that they are, on our data set, better predictors than "traditional" code metrics, which can only be collected at a later phase of the software development processes.

Key-words: Object-Oriented Design Metrics; Error Prediction Model; Object-Oriented Software Development; C++ Programming Language.

* V. Basili and W. Melo are with the University of Maryland, Institute for Advanced Computer Studies and Computer Science Dept., A. V. Williams Bldg., College Park, MD 20742 USA. {basili | melo}@cs.umd.edu

L. Briand is with the CRIM, 1801 McGill College Av., Montréal (Québec), H3A 2N4, Canada. lbriand@crim.ca

Evaluation of the prediction power of CK Metrics

WMC

DIT

CBO

NOC

LCOM

RFC

1996

CK Metrics

Metric	Description
WMC	Weighted Methods per Class
DIT	Depth of Inheritance Tree
NOC	Number of Children
CBO	Coupling Between Objects
RFC	Response for a class
LCOM	Lack Of Cohesion of Methods

CK Metrics

Metric	Description
WMC	**Weighted Methods per Class**
DIT	Depth of Inheritance Tree
NOC	Number of Children
CBO	Coupling Between Objects
RFC	Response for a class
LCOM	Lack Of Cohesion of Methods

WMC is somewhat relevant

CK Metrics

Metric	Description
WMC	Weighted Methods per Class
DIT	**Depth of Inheritance Tree**
NOC	**Number of Children**
CBO	**Coupling Between Objects**
RFC	**Response for a class**
LCOM	Lack Of Cohesion of Methods

Some metrics are good predictors

CK Metrics

Metric	Description
WMC	Weighted Methods per Class
DIT	Depth of Inheritance Tree
NOC	Number of Children
CBO	Coupling Between Objects
RFC	Response for a class
LCOM	**Lack Of Cohesion of Methods**

LCOM has a low predictive power

Research provides different alternatives

- Structural metrics
- Complexity metrics
- Process metrics
- Developer-based metrics
- Textual metrics

McCabe Cyclomatic Complexity

Correlation between complexity and number of bugs

$$V(G) = \#edge - \#node + 2$$

Correlation Analysis

1970s

Halstead Complexity Metrics

Metrics based on the number of operators and operands

$$Volume = N * \log_2 n$$

$$\#\ defects = Volume\ /\ 3000$$

Correlation Analysis

1970s

Complexity Metrics

Predicting Defects for Eclipse

Thomas Zimmermann
Saarland University
tz@acm.org

Rahul Premraj
Saarland University
premraj@cs.uni-sb.de

Andreas Zeller
Saarland University
zeller@acm.org

Abstract

We have mapped defects from the bug database of Eclipse (one of the largest open-source projects) to source code locations. The resulting data set lists the number of pre- and post-release defects for every package and file in the Eclipse releases 2.0, 2.1, and 3.0. We additionally annotated the data with common complexity metrics. All data is publicly available and can serve as a benchmark for defect prediction models.

1. Introduction

Why is it that some programs are more failure-prone than others? This is one of the central questions of software engineering. To answer it, we must first know *which* programs are more failure-prone than others. With this knowledge, we can search for properties of the program or its development process that commonly correlate with defect density; in other words, once we can measure the effect, we can search for its causes.

One of the most abundant, widespread, and reliable sources for failure information is a *bug database,* listing all the problems that occurred during the software lifetime. Unfortunately, bug databases frequently do not directly record how, where, and by whom the problem in question was fixed. This information is hidden in the *version database,* recording all changes to the software source code.

In recent years, a number of techniques have been developed to relate bug reports to fixes [3, 6, 17]. Since we thus can relate bugs to fixes, and fixes to the locations they apply to, we can easily determine the *number of defects* of a component—simply by counting the applied fixes.

We have conducted such a work on the code base of the Eclipse programming environment. In particular, we have computed the mapping of packages and classes to the number of defects that were reported in the first six months *before* and *after* release. In previous work, we made our *Eclipse bug data set* freely

Project:	Eclipse (eclipse.org)
Content:	Defect counts (pre- and post-release) Complexity metrics
Releases:	2.0, 2.1, and 3.0
Level:	Packages and files
URL:	http://www.st.cs.uni-sb.de/softevo/bug-data/eclipse
More data:	Eclipse source code (for archived releases): http://archive.eclipse.org/eclipse/downloads/

Figure 1. Summary of our data set.

available [15]. For this paper, we extended our data with common complexity metrics and the counts of syntactic elements (obtained from abstract syntax trees). With this new data, many predictor models can be built out of the box which we demonstrate in this paper.

We invite readers to use our data for research purposes and to build their own models. We hope that the public availability of data sets like ours will foster empirical research in software engineering, just like the public availability of open source programs fostered research in program analysis.

2. State of the art

Predicting which components are more failure-prone than others has been addressed by a number of researchers in the past. This work, discussed below, used either complexity metrics or historical data to predict failures.

2.1. Complexity metrics

Typically, research on defect-proneness defines metrics to capture the complexity of software and builds models that relate these metrics to defect-proneness [4]. Basili et al. [1] were among the first to validate that OO metrics are useful for predicting defect density. Subramanyam and Krishnan [18] presented a survey on eight more empirical studies, all showing that OO metrics are significantly associated with defects. Post-release defects are the defects that actually matter for the end-users of a program. Only few studies ad-

Third International Workshop on Predictor Models in Software Engineering (PROMISE'07)
0-7695-2954-2/07 $20.00 © 2007 IEEE

Evaluation of the prediction power of complexity metrics

Complexity metrics at methods, class, and file level

These metrics correlate with defects in Eclipse

2007

Research provides different alternatives

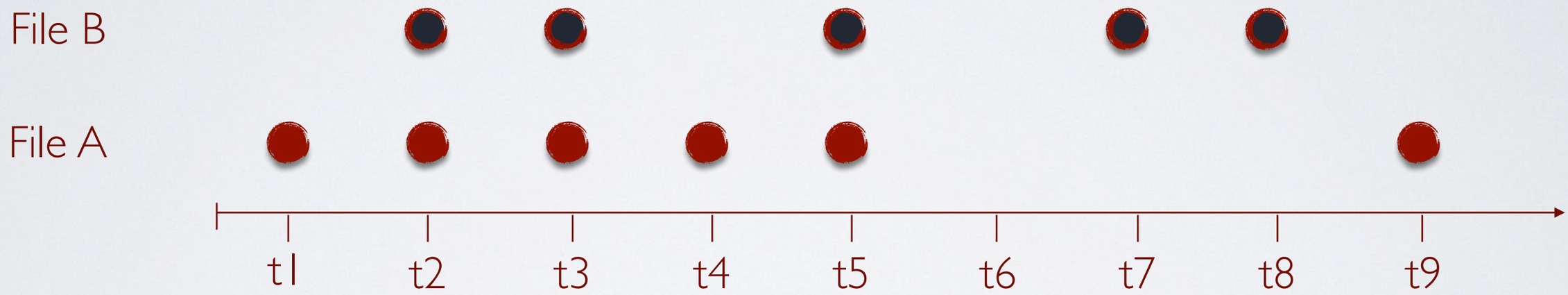
- Structural metrics
- Complexity metrics
- Process metrics
- Developer-based metrics
- Textual metrics

Process Metrics

Measuring characteristics of the development process

Process Metrics

Measuring characteristics of the development process

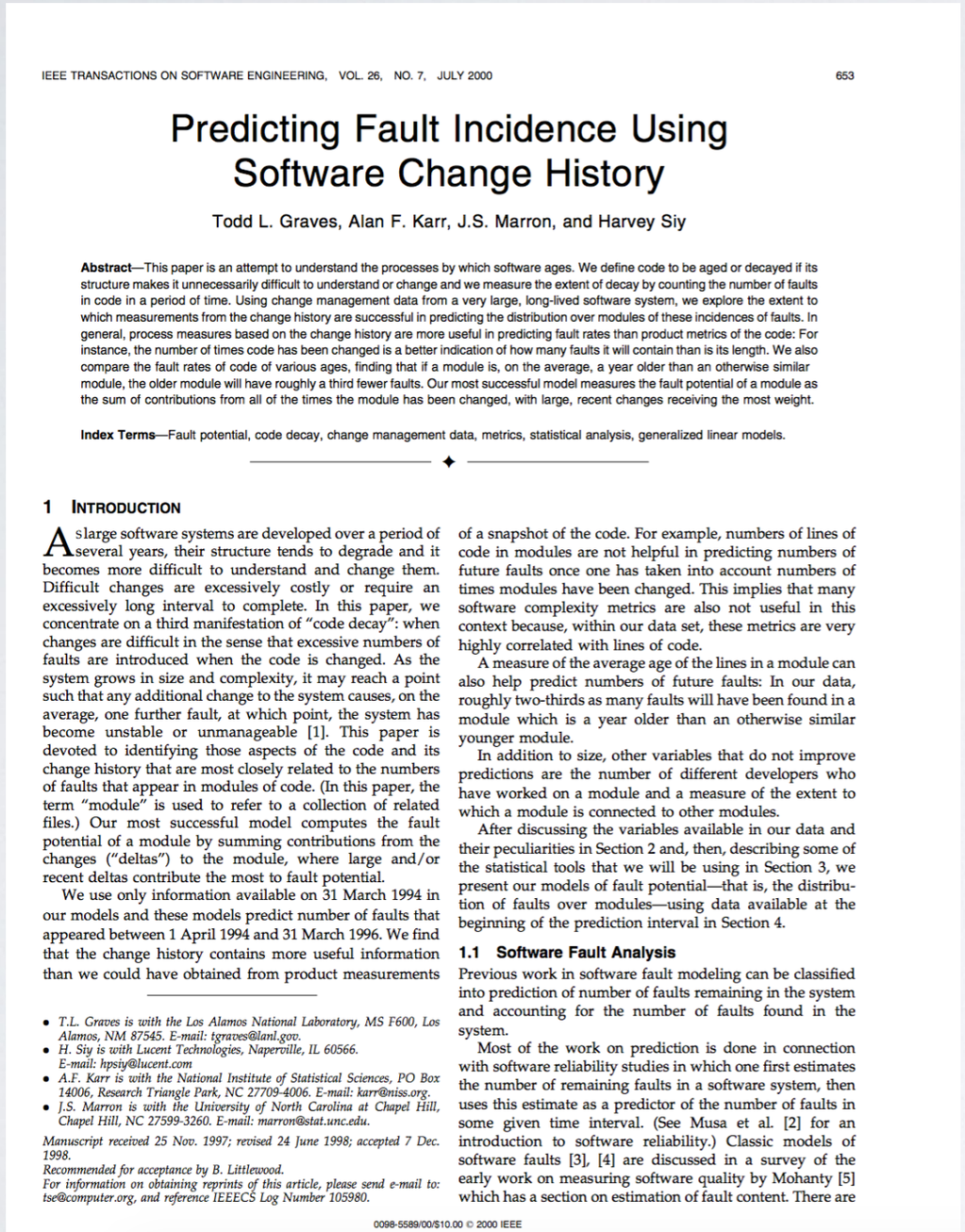


Number of Changes

Commit Size

Entropy of Changes

Process Metrics



Predicting Fault Incidence Using Software Change History

Todd L. Graves, Alan F. Karr, J.S. Marron, and Harvey Siy

Abstract—This paper is an attempt to understand the processes by which software ages. We define code to be aged or decayed if its structure makes it unnecessarily difficult to understand or change and we measure the extent of decay by counting the number of faults in code in a period of time. Using change management data from a very large, long-lived software system, we explore the extent to which measurements from the change history are successful in predicting the distribution over modules of these incidences of faults. In general, process measures based on the change history are more useful in predicting fault rates than product metrics of the code: For instance, the number of times code has been changed is a better indication of how many faults it will contain than is its length. We also compare the fault rates of code of various ages, finding that if a module is, on the average, a year older than an otherwise similar module, the older module will have roughly a third fewer faults. Our most successful model measures the fault potential of a module as the sum of contributions from all of the times the module has been changed, with large, recent changes receiving the most weight.

Index Terms—Fault potential, code decay, change management data, metrics, statistical analysis, generalized linear models.

1 Introduction

As large software systems are developed over a period of several years, their structure tends to degrade and it becomes more difficult to understand and change them. Difficult changes are excessively costly or require an excessively long interval to complete. In this paper, we concentrate on a third manifestation of "code decay": when changes are difficult in the sense that excessive numbers of faults are introduced when the code is changed. As the system grows in size and complexity, it may reach a point such that any additional change to the system causes, on the average, one further fault, at which point, the system has become unstable or unmanageable [1]. This paper is devoted to identifying those aspects of the code and its change history that are most closely related to the numbers of faults that appear in modules of code. (In this paper, the term "module" is used to refer to a collection of related files.) Our most successful model computes the fault potential of a module by summing contributions from the changes ("deltas") to the module, where large and/or recent deltas contribute the most to fault potential.

We use only information available on 31 March 1994 in our models and these models predict number of faults that appeared between 1 April 1994 and 31 March 1996. We find that the change history contains more useful information than we could have obtained from product measurements of a snapshot of the code. For example, numbers of lines of code in modules are not helpful in predicting numbers of future faults once one has taken into account numbers of times modules have been changed. This implies that many software complexity metrics are also not useful in this context because, within our data set, these metrics are very highly correlated with lines of code.

A measure of the average age of the lines in a module can also help predict numbers of future faults: In our data, roughly two-thirds as many faults will have been found in a module which is a year older than an otherwise similar younger module.

In addition to size, other variables that do not improve predictions are the number of different developers who have worked on a module and a measure of the extent to which a module is connected to other modules.

After discussing the variables available in our data and their peculiarities in Section 2 and, then, describing some of the statistical tools that we will be using in Section 3, we present our models of fault potential—that is, the distribution of faults over modules—using data available at the beginning of the prediction interval in Section 4.

1.1 Software Fault Analysis

Previous work in software fault modeling can be classified into prediction of number of faults remaining in the system and accounting for the number of faults found in the system.

Most of the work on prediction is done in connection with software reliability studies in which one first estimates the number of remaining faults in a software system, then uses this estimate as a predictor of the number of faults in some given time interval. (See Musa et al. [2] for an introduction to software reliability.) Classic models of software faults [3], [4] are discussed in a survey of the early work on measuring software quality by Mohanty [5] which has a section on estimation of fault content. There are

- *T.L. Graves is with the Los Alamos National Laboratory, MS F600, Los Alamos, NM 87545. E-mail: tgraves@lanl.gov.*
- *H. Siy is with Lucent Technologies, Naperville, IL 60566. E-mail: hpsiy@lucent.com*
- *A.F. Karr is with the National Institute of Statistical Sciences, PO Box 14006, Research Triangle Park, NC 27709-4006. E-mail: karr@niss.org.*
- *J.S. Marron is with the University of North Carolina at Chapel Hill, Chapel Hill, NC 27599-3260. E-mail: marron@stat.unc.edu.*

Manuscript received 25 Nov. 1997; revised 24 June 1998; accepted 7 Dec. 1998.
Recommended for acceptance by B. Littlewood.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 105980.

Testing of both product and process metrics

Product metrics are poor indicators of bugs!

Number of Changes are better than product metrics

2000

Bug-caching

Analysis of the Reliability of a Subset of Change Metrics for Defect Prediction

Raimund Moser
Free University of Bolzano-Bozen
Piazza Domenicani 3
I-39100 Bolzano, Italy
+39 0471016138
raimund.moser@unibz.it

Witold Pedrycz
University of Alberta
T6G 2V4 Edmonton
Alberta, Canada
+1 7804923333
pedrycz@ee.ualberta.ca

Giancarlo Succi
Free University of Bolzano-Bozen
Piazza Domenicani 3
I-39100 Bolzano, Italy
+39 0471016130
giancarlo.succi@unibz.it

ABSTRACT

In this paper, we describe an experiment, which analyzes the relative importance and stability of change metrics for predicting defects for 3 releases of the Eclipse project. The results indicate that out of 18 change metrics 3 metrics contain most information about software defects. Moreover, those 3 metrics remain stable across 3 releases of the Eclipse project. A comparative analysis with the full model shows that the prediction accuracy is not too much affected by using a subset of 3 metrics and the recall even improves.

Categories and Subject Descriptors

D.2.8 [**Metrics**]: Process metrics and product metrics. D.2.9 [**Management**]: Software quality assurance.

General Terms

Measurement, Experimentation.

Keywords

Defect prediction, software metrics, feature selection.

1. INTRODUCTION

Although significant research has been conducted in the area of defect prediction the results obtained so far are not conclusive: we still do not have empirical or theoretical evidence of which are the "best" attributes for defect prediction. It seems that attributes referring to the development process and environment and to the change history of source code are better defect predictors than static code attributes [4] [2] [6]. Moreover, "best sets" of predictors appear to depend rather on the characteristics of single data sets than on some underlying defect generation mechanism [3]. Nevertheless, it is important to explore what are the predictors, which contain most information about defects in source code: first, such knowledge helps to reduce the effort for collecting metrics and for building and customizing models. Second, it can be used as starting point for a future theory of software defects.

ESEM'08, October 9–10, 2008, Kaiserslautern, Germany.

In this research we analyze the relationship between change metrics and defects for 3 releases of the Eclipse project. We start from our findings of a previous study, which we use as basis for this work: for the Eclipse project we found change metrics to be better defect predictors than static code attributes and a cost-sensitive decision tree learner a suitable model [4]. Based on these findings we pose the following research questions:

- Which change metrics contain most information of software defects and are stable across different releases of a project?
- How does the accuracy of models based on a reduced set of predictors compare to the one of the full model?

The remaining of this paper is structured as follows: Section 2 describes the experimental set-up and data; Section 3 presents the results and their discussion. Finally, in Section 4 we draw the conclusions and identify future research needs.

2. EXPERIMENTAL SET-UP

For our experiment we use the Eclipse bug data set, which has been published on the PROMISE repository by Zimmermann *et al.* [7]. It contains for each file of 3 releases of the Eclipse project: 198 static code attributes and the number of pre- and post release defects. We add 18 change metrics extracted from the Eclipse CVS repository skipping the subset of original files, which do not show a complete CVS history. Table 1 shows a summary of the data set used in this study and the class distribution, i.e., the percentage of defective files. The percentage in brackets in column 2 of Table 1 indicates the size of our data set with respect to the original one.

Table 1: Summary of the data.

Release	Number of files	No defects	Defective
2.0	3851 (57%)	2665	1186
2.1	5341 (68%)	4087	1254
3.0	5347 (81%)	3622	1725

The 18 change metrics we extract from the code repository of the Eclipse project are reported in Table 2. In our previous work [4] we showed that those 18 change metrics outperform models based

A comprehensive study on product and process metrics

Past faults and number of changes are the better indicators of bug presence

2008

Bug-caching

The higher the **number of previous bugs,** the higher the **bug-proneness** of a code component

2008

Entropy of Changes

Predicting Faults Using the Complexity of Code Changes

Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University, Canada
ahmed@cs.queensu.ca

Abstract

Predicting the incidence of faults in code has been commonly associated with measuring complexity. In this paper, we propose complexity metrics that are based on the code change process instead of on the code. We conjecture that a complex code change process negatively affects its product, i.e., the software system. We validate our hypothesis empirically through a case study using data derived from the change history for six large open source projects. Our case study shows that our change complexity metrics are better predictors of fault potential in comparison to other well-known historical predictors of faults, i.e., prior modifications and prior faults.

1 Introduction

Managing the complexity of a project is a paramount goal while striving to meet user needs. The literature contains a wealth of metrics (*e.g.* [19]) which measure the complexity of the source code. However, little attention has been paid to measuring and controlling the complexity of the code change process. This process plays a central role in a project since it is responsible for producing the code needed to satisfy requirements, while dealing with the complexities and challenges associated with the current code base and other facets of the project such as its design, customer requirements, the team structure and size, market pressure, and problem domain. A software system with a complex code change process is undesirable since it will likely produce a system which has many faults and the project will face delays.

Four lines of prior work motivate our intuition about the importance of the code change process and historical code changes in predicting the incidence of faults:

1. Studies by Briand *et al.* [2], Graves *et al.* [11], Khoshgoftaar *et al.* [20], Leszak *et al.* [22], and Nagappan and Ball [26] indicate that prior modifications to a file are a good predictor of its fault potential (*i.e., the more a file is changed, the more likely it will contain faults*).
2. Studies by Graves *et al.* [11] and Leszak *et al.* [22], on commercial systems, and recently by Herraiz *et al.* [18] on open source systems show that most code complexity metrics highly correlate with LOC, a much simpler metric.
3. Studies, such as the one by Moser *et al.* [25], show that process metrics outperform code metrics as predictors of future faults.
4. Studies, such as the one by Yu *et al.* [37], indicate that prior faults are good predictors of future faults.

In prior work, we used concepts from information theory to define change complexity models which capture our intuition about complex changes. Events such as large refactorings or release delays were accompanied with increases in our proposed model measurements [14, 15]. Our earlier results lead us to the following conjecture:

> *A complex code change process negatively affects its product, the software system. The more complex changes to a file, the higher the chance the file will contain faults.*

In this paper, we extend our change complexity models and study the ability of our proposed model measurements to predict the incidence of faults in a software system. In particular, we compare the performance of predictors based on our complexity models with predictors based on the number of prior modifications and prior faults. Based on a case study using six large open source projects, our results indicate that our change complexity models are better predictors of fault potential in contrast to other historical predictors (such as prior modifications and prior faults).

Overview Of Paper. This paper is organized as follows. Section 2 gives our view of the code change process. Section 3 present Shannon's entropy which we use to quantify the complexity of code changes. Sections 4, 5, and 6 present the complexity models we use in our work. Section 4 introduces our first and simplest model for the complexity of code changes – **The Basic Code Change (BCC) Model**. We proceed to give a more elaborate and complete model in Section 5 – **The Extended Code Change (ECC) Model**.

Exploiting the complexity of the change process

The more changes are distributed, the higher the entropy of the process

2009

Entropy of Changes

Feature Introduction (FI)

Changes applied to add or enhance features. They are used to measure the complexity of change process.

Fault Repairing Modification (FR)

Changes applied to fix bugs. They are used to validate the model.

General Maintenance Modification (GM)

Other modifications, such as copyright updates or re-indentation of source code.

2009

Entropy of Changes

Feature Introduction (FI)

Changes applied to add or enhance features. They are used to measure the complexity of change process.

Fault Repairing Modification (FR)

Changes applied to fix bugs. They are used to validate the model.

General Maintenance Modification (GM)

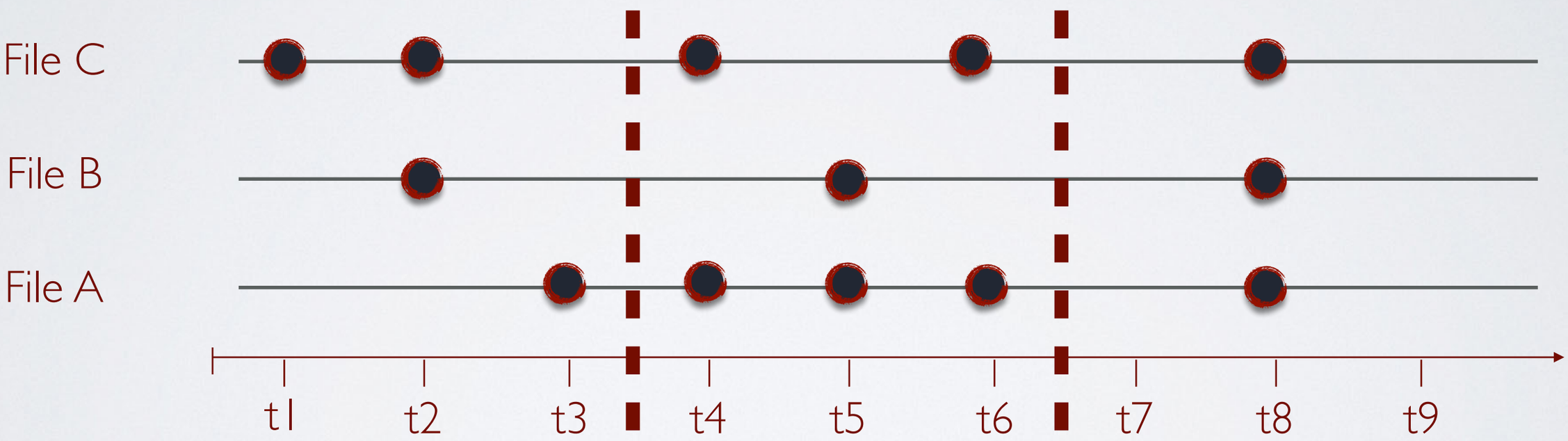
Other modifications, such as copyright updates or re-indentation of source code.

2009

Entropy of Changes

Shannon Entropy

$$H = -\sum_{i=1}^{M} P_i \log_2 P_i$$



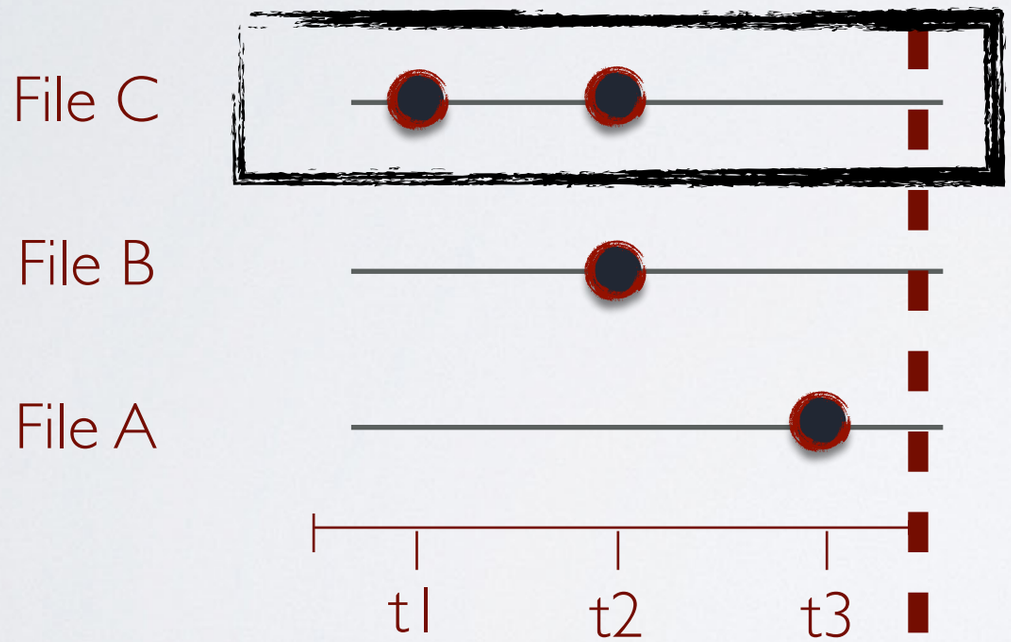
2009

Entropy of Changes

Shannon Entropy

$$H = -\sum_{i=1}^{M} P_i \log_2 P_i$$

Period 1



H(C)= - 2/4 * log(2/4)

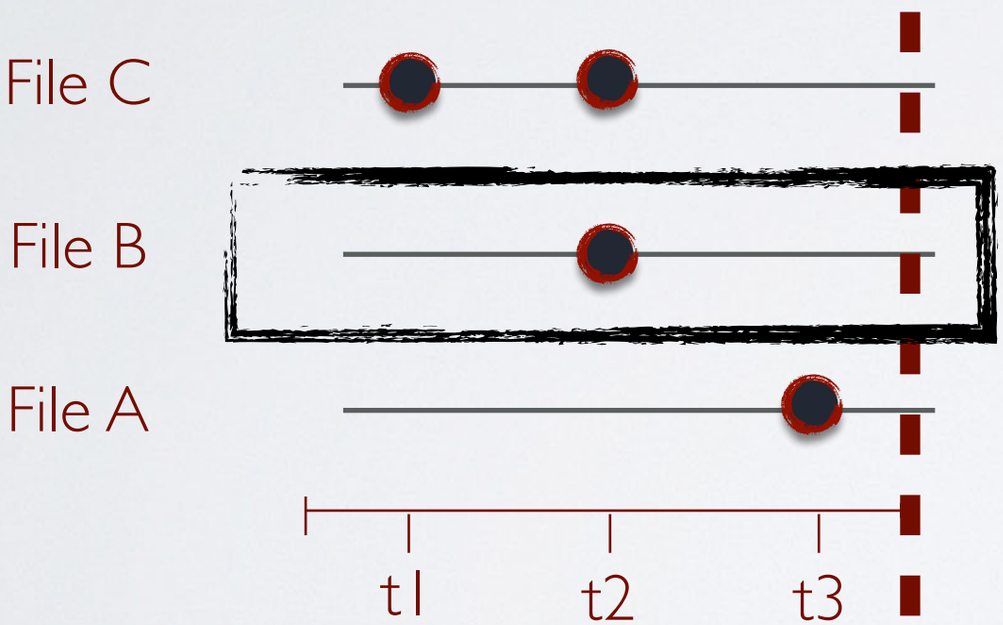
2009

Entropy of Changes

Shannon Entropy

$$H = -\sum_{i=1}^{M} P_i \log_2 P_i$$

Period 1



H(B)= - 1/4 * log(1/4)

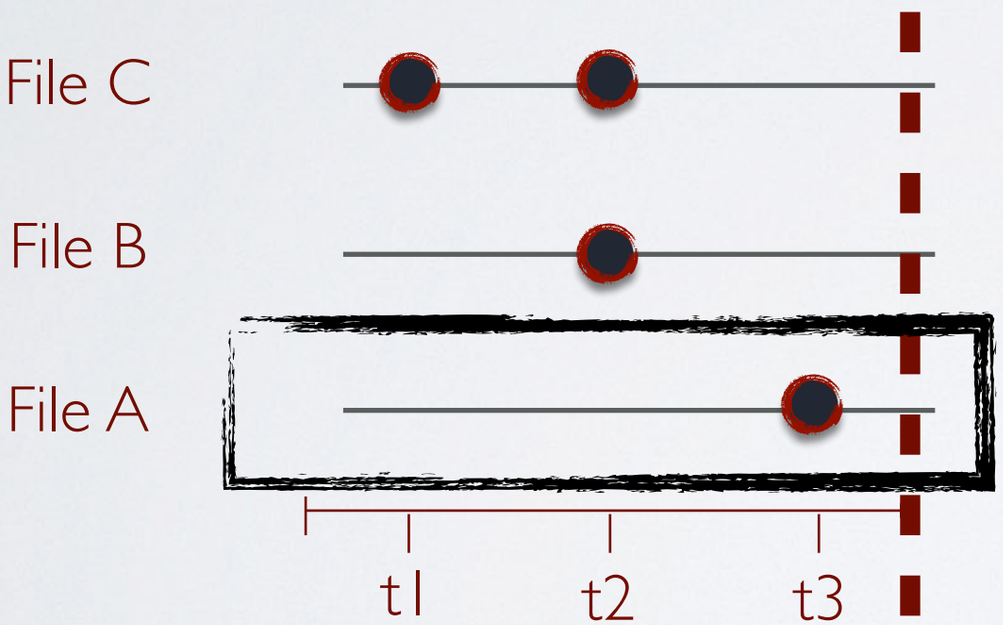
2009

Entropy of Changes

Shannon Entropy

$$H = -\sum_{i=1}^{M} P_i \log_2 P_i$$

Period 1



H(A)= - 1/4 * log(1/4)

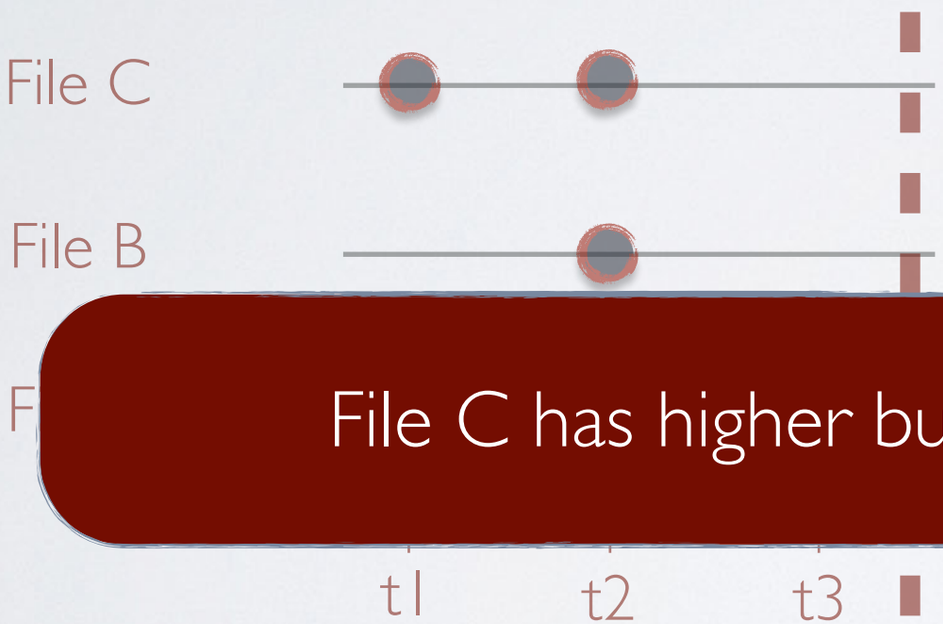
2009

Entropy of Changes

Shannon Entropy

$$H = -\sum_{i=1}^{M} P_i \log_2 P_i$$

Period I



H(C)= - 2/4 * log(2/4)

H(B)= - 1/4 * log(1/4)

File C has higher bug-proneness than files A and B

2009

Entropy of Changes

Models based on **entropy** of changes are **better** bug predictors than number of changes or faults

2009

Research provides different alternatives

Structural metrics

Complexity metrics

Process metrics

Developer-based metrics

Textual metrics

Number of Developers

Empir Software Eng
DOI 10.1007/s10664-011-9178-4

The limited impact of individual developer data on software defect prediction

Robert M. Bell · Thomas J. Ostrand · Elaine J. Weyuker

Editors: Tim Menzies and Gunes Koru

Abstract Previous research has provided evidence that a combination of static code metrics and software history metrics can be used to predict with surprising success which files in the next release of a large system will have the largest numbers of defects. In contrast, very little research exists to indicate whether information about individual developers can profitably be used to improve predictions. We investigate whether files in a large system that are modified by an individual developer consistently contain either more or fewer faults than the average of all files in the system. The goal of the investigation is to determine whether information about which particular developer modified a file is able to improve defect predictions. We also extend earlier research evaluating use of counts of the number of developers who modified a file as predictors of the file's future faultiness. We analyze change reports filed for three large systems, each containing 18 releases, with a combined total of nearly 4 million LOC and over 11,000 files. A *buggy file ratio* is defined for programmers, measuring the proportion of faulty files in Release R out of all files modified by the programmer in Release R-1. We assess the consistency of the buggy file ratio across releases for individual programmers both visually and within the context of a fault prediction model. Buggy file ratios for individual programmers often varied widely across all the releases that they participated in. A prediction model that takes account of the history of faulty files that were changed by individual developers shows improvement over the standard negative binomial

R. M. Bell · T. J. Ostrand (✉) · E. J. Weyuker
AT&T Labs - Research, 180 Park Avenue, Florham Park, NJ 07932, USA
e-mail: ostrand@research.att.com

R. M. Bell
e-mail: rbell@research.att.com

E. J. Weyuker
e-mail: weyuker@research.att.com

Published online: 17 September 2011

Exploiting the number of developers working on a file

The higher the number of developers that touch a file, the higher its bug-proneness

This metric partially positively impact the performances of process and product models

2013

Developers' Scattered Changes

On the Role of Developer's Scattered Changes in Bug Prediction

Dario Di Nucci*, Fabio Palomba*, Sandro Siravo†, Gabriele Bavota‡, Rocco Oliveto†, Andrea De Lucia*
*University of Salerno, Fisciano (SA), Italy — †University of Molise, Pesche (IS), Italy
‡Free University of Bozen-Bolzano, Bolzano (BZ), Italy

***Abstract*—The importance of human-related factors in the introduction of bugs has recently been the subject of a number of empirical studies. However, these observations have not been captured yet in bug prediction models which simply exploit product metrics or process metrics based on the number and type of changes or on the number of developers working on a software component. Some previous studies have demonstrated that focused developers are less prone to introduce defects than non focused developers. According to this observation, software components changed by focused developers should also be less error prone than software components changed by less focused developers. In this paper we capture this observation by measuring the structural and semantic scattering of changes performed by the developers working on a software component and use these two measures to build a bug prediction model. Such a model has been evaluated on five open source systems and compared with two competitive prediction models: the first exploits the number of developers working on a code component in a given time period as predictor, while the second is based on the concept of code change entropy. The achieved results show the superiority of our model with respect to the two competitive approaches, and the complementarity of the defined scattering measures with respect to standard predictors commonly used in the literature.**

I. Introduction

Empirical studies have been carried out to assess under which circumstances and during which coding activities developers tend to introduce bugs (see e.g., [1], [2], [3], [4], [5], [6]). Also, bug prediction techniques built on top of *process metrics* (i.e., metrics capturing specific aspects of the development process, like the frequency of changes performed to code components) have been proposed [7], [8], [9], [10], [11], [12], [13]. Several of these techniques have demonstrated their superiority [14] with respect to approaches only exploiting *product metrics* (i.e., metrics capturing intrinsic characteristics of the code components, like their size and complexity) [15], [16], [17], [18], [19].

However, the role of developer-related factors in the bug prediction field is only partially explored. Indeed, our knowledge on this topic is mainly amenable to few empirical studies performed in the last years. The first is the one by Eyolfson *et al.* [2], who showed that more experienced developers tend to introduce less faults in software systems. The second has been performed by Rahman and Devanbu [3], and contradicts in part the study by Eyolfson *et al.* [2] by showing that the experience of a developer has no clear association with the introduction of buggy code. Bird *et al.* [20] found that high levels of ownership are associated with fewer bugs. Posnett *et al.* [6] showed that focused developers (i.e., developers focusing their attention on a specific part of the system) introduce fewer bugs than unfocused developers. This observation has been confirmed by Tufano *et al.* [21], who reported as commits (i.e., code changes) impacting unrelated code components (i.e., files belonging to different subsystems or, more in general, implementing unrelated responsibilities) are more likely to introduce bugs with respect to commits performed on highly related code components. These works have pioneered the investigation of human-related factors in the context of bug introduction.

Although such studies showed the importance of human-related factors in bug prediction, these observations have not been captured yet in bug prediction models based on process metrics extracted from version history. Indeed, previous work have proposed the use of predictors based (i) on the number of developers working on a code component [10] [11]; (ii) on the analysis of change-proneness [14] [12] [13]; and (iii) on the entropy of changes [9]. None of them consider how focused are the developers performing changes and how scattered are these changes. With this work we aim at making a further step ahead, by studying the role played by *scattering changes* in bug prediction. We firstly define two measures, namely the developer's *structural* and *semantic scattering*. The first aims at assessing how "structurally far" in the software project are the code components modified in a given time period by a developer. The "structural distance" between two code components is measured as the number of subsystems one needs to cross in order to reach one component from the other. The second measure (i.e., the *semantic scattering*) is instead meant to capture how much spread in terms of implemented responsibilities are the code components modified in a given time period by a developer. We expect that high levels of *structural* and *semantic scattering* make the developer more error-prone. In order to verify our conjecture we build two predictors exploiting the proposed measures, and then we use them in a prediction model, comparing its performances with two baseline techniques based on the number of developers working on a code component and the entropy of changes, respectively [10], [9].

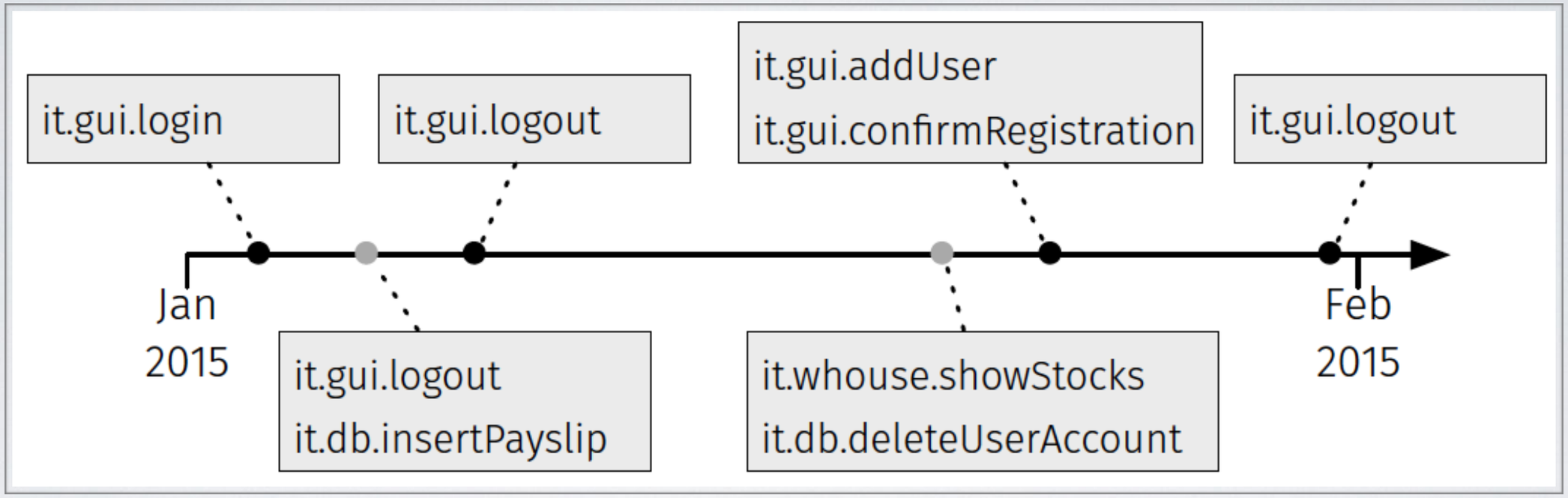
The context of our empirical investigation is represented by five large Java open-source systems. The results achieved show the superiority of our model, achieving a prediction accuracy ranging between 68% and 94%, as compared to the 43%-74% achieved by the change entropy model [9] and the 19%-49% obtained by exploiting the number of developers working on a code component as predictor.

Exploiting how developers apply changes over the system

Focused developers tend to be less bug-prone than non-focused developers

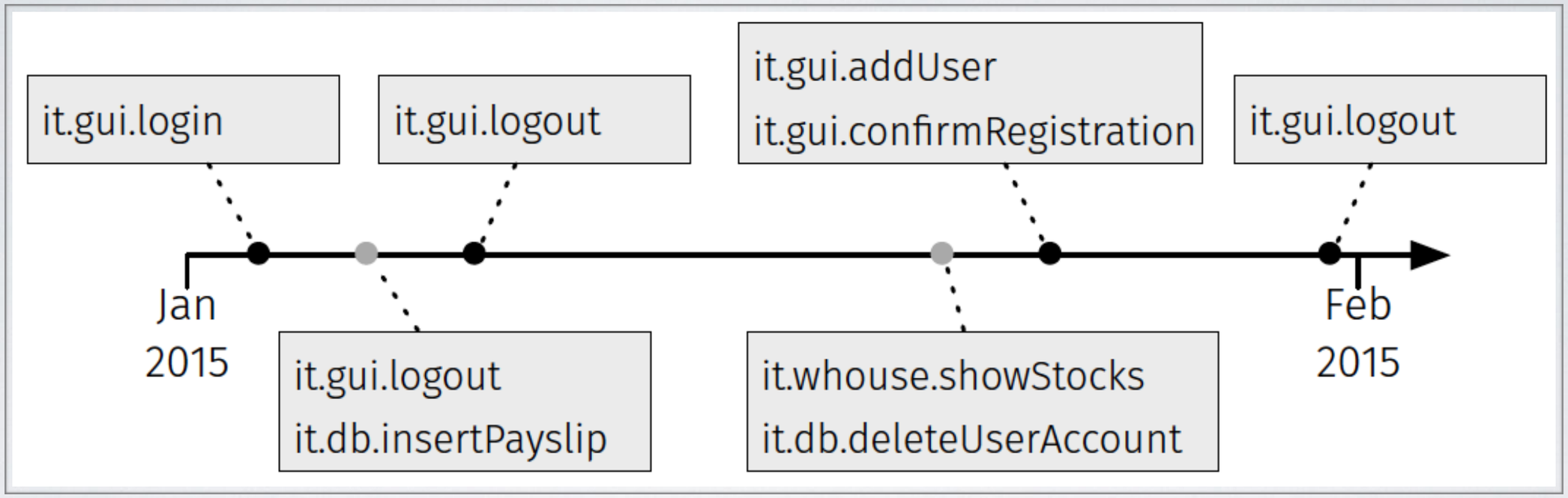
2015

Developers' Scattered Changes



2015

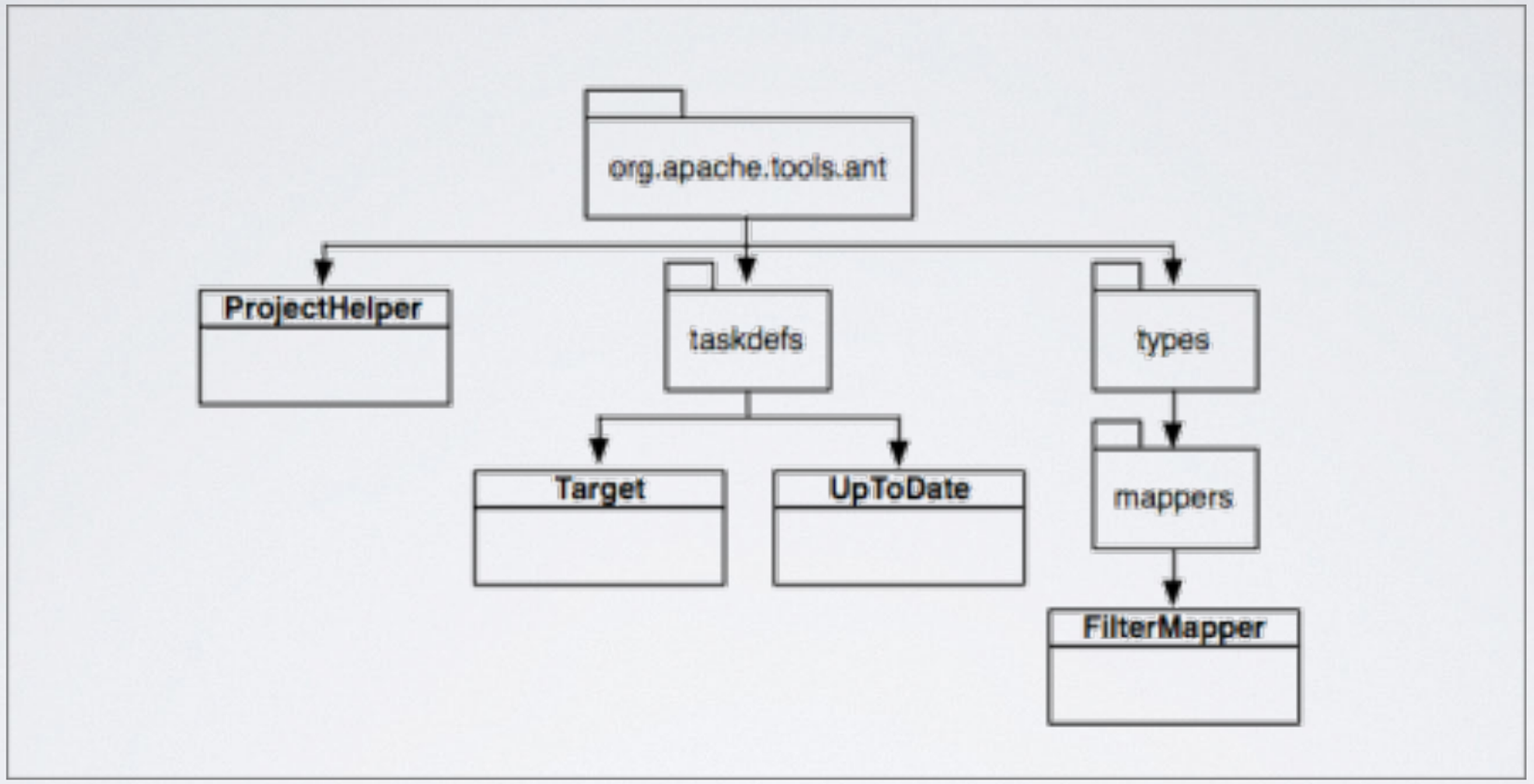
Developers' Scattered Changes



The second developer is less focused because she makes more scattering changes

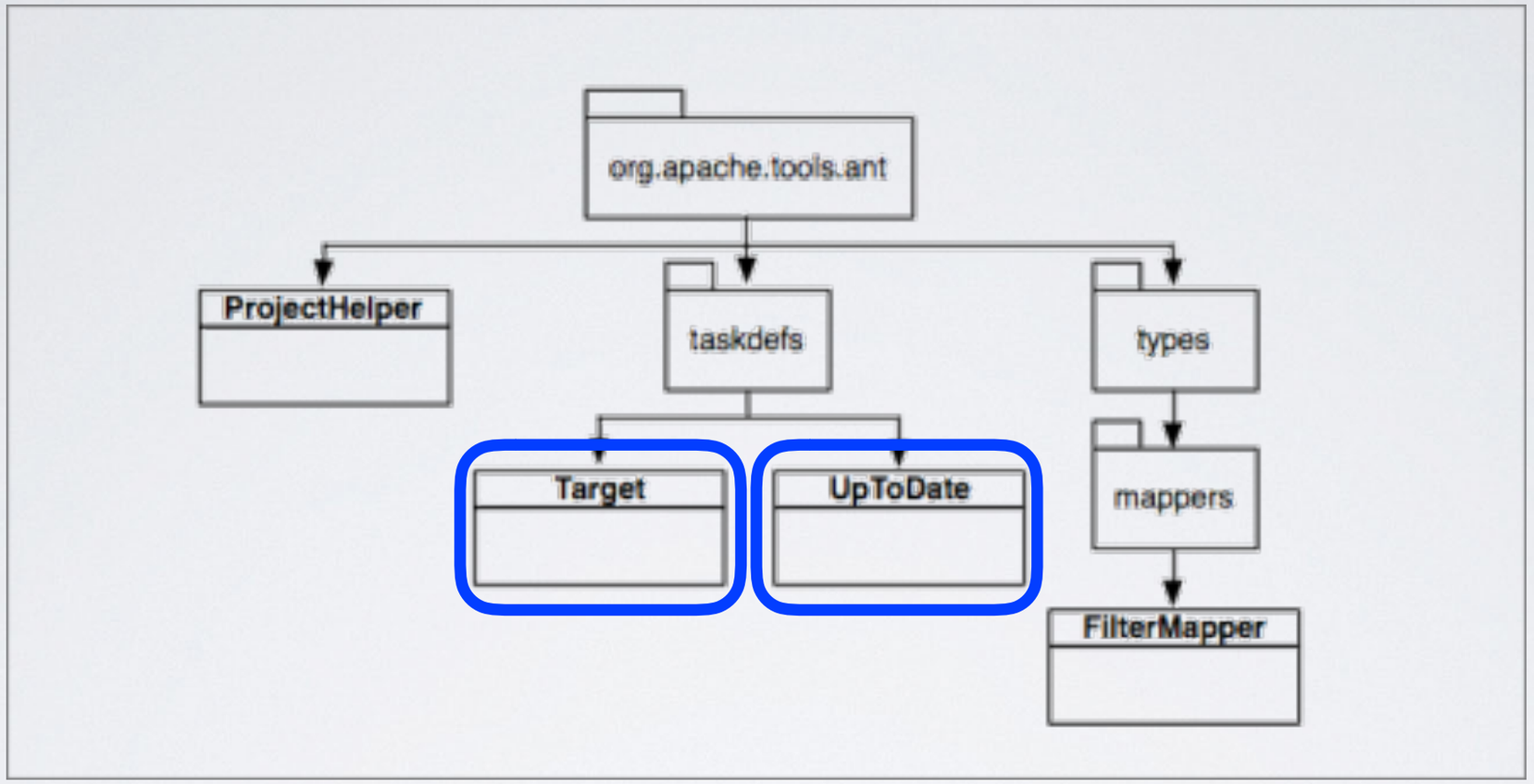
2015

Developers' Scattered Changes



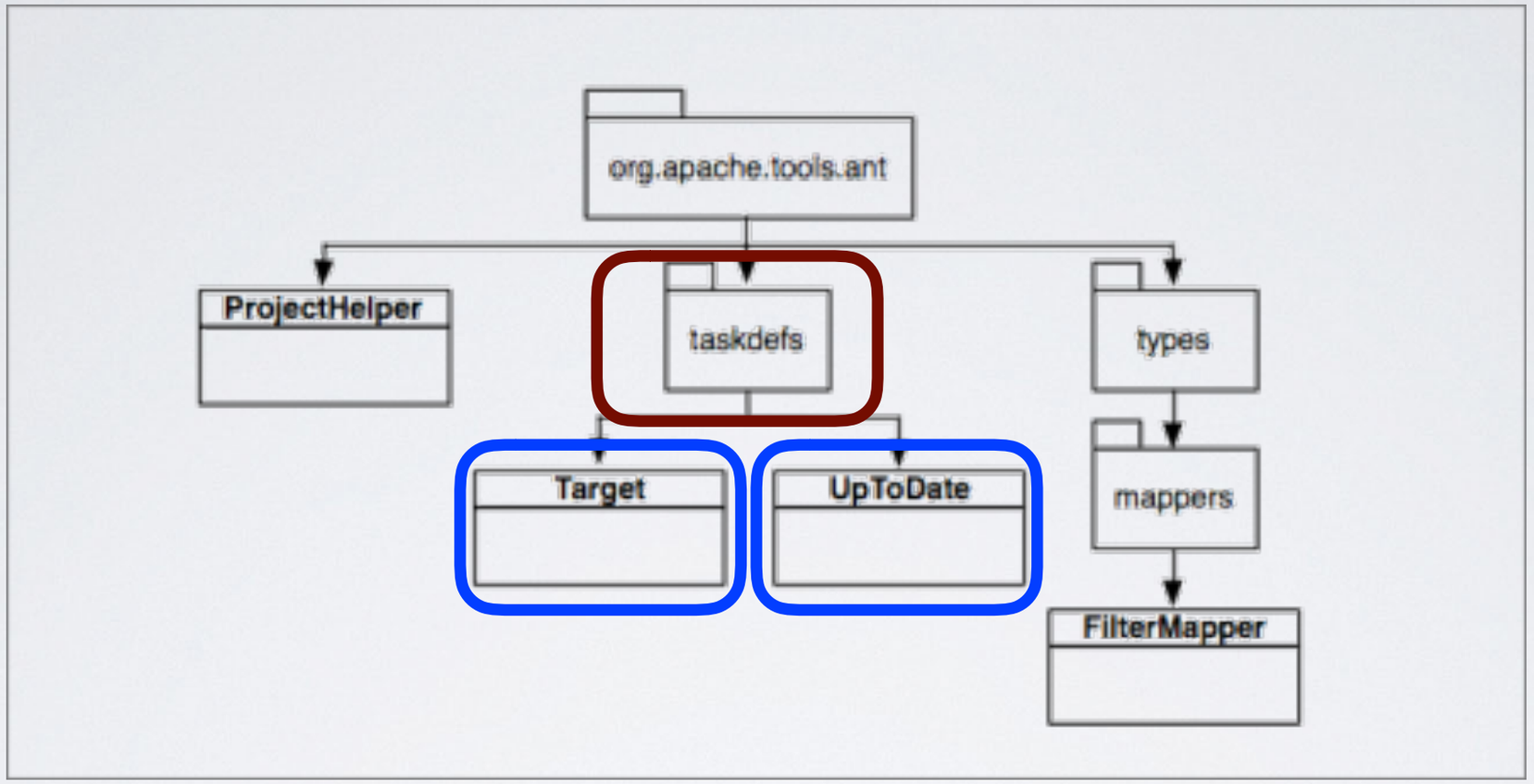
2015

Developers' Scattered Changes



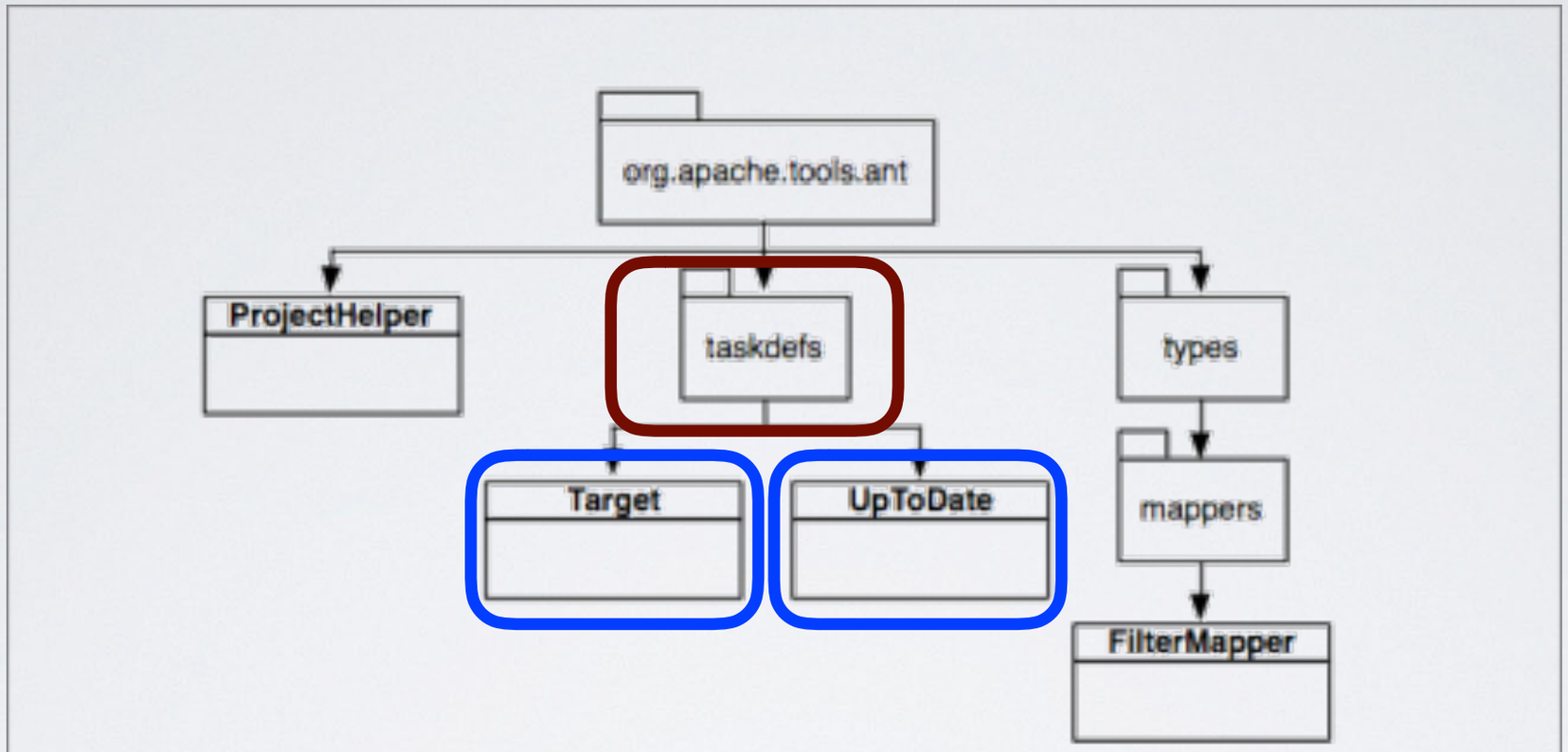
2015

Developers' Scattered Changes



2015

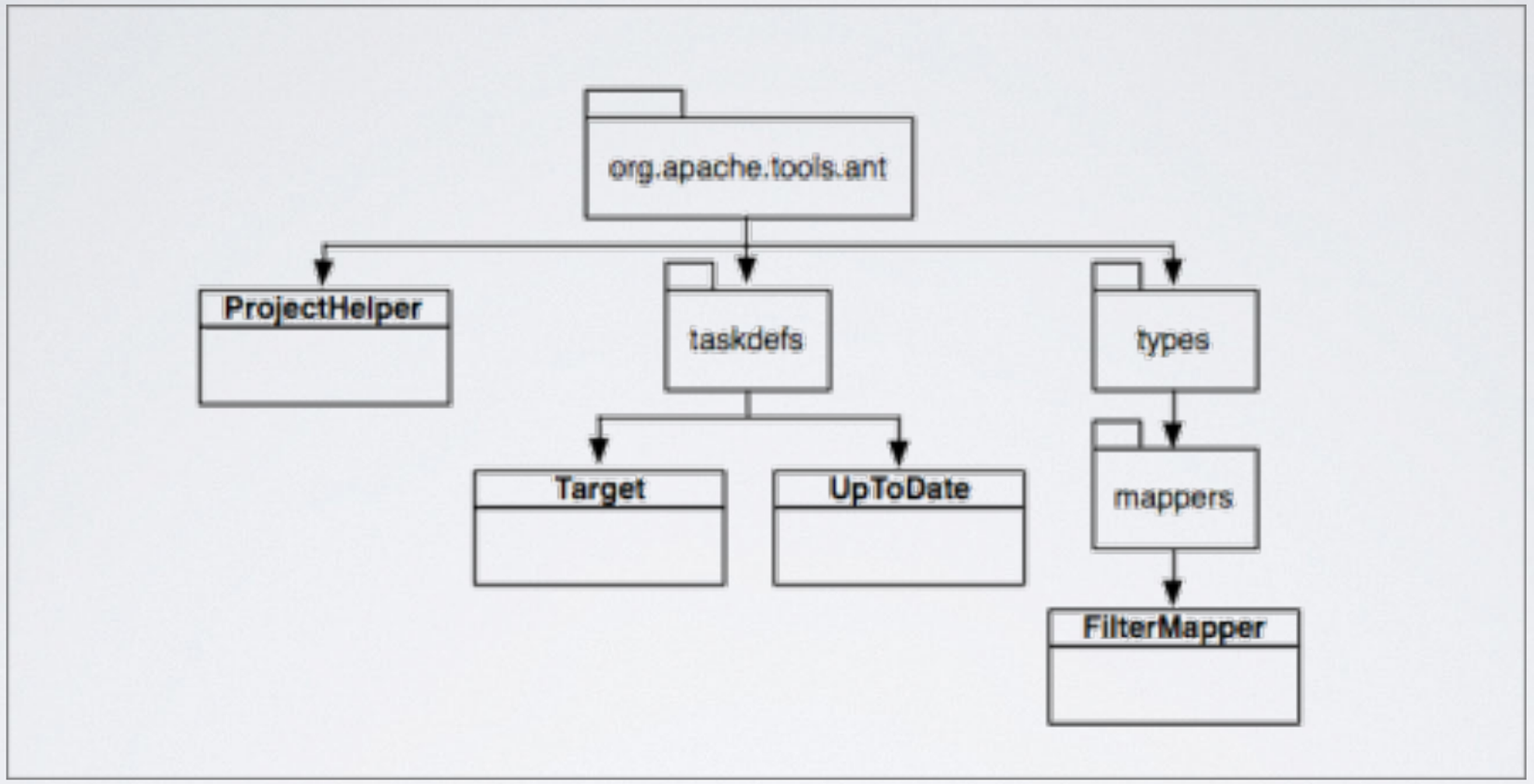
Developers' Scattered Changes



The structural distance is 0

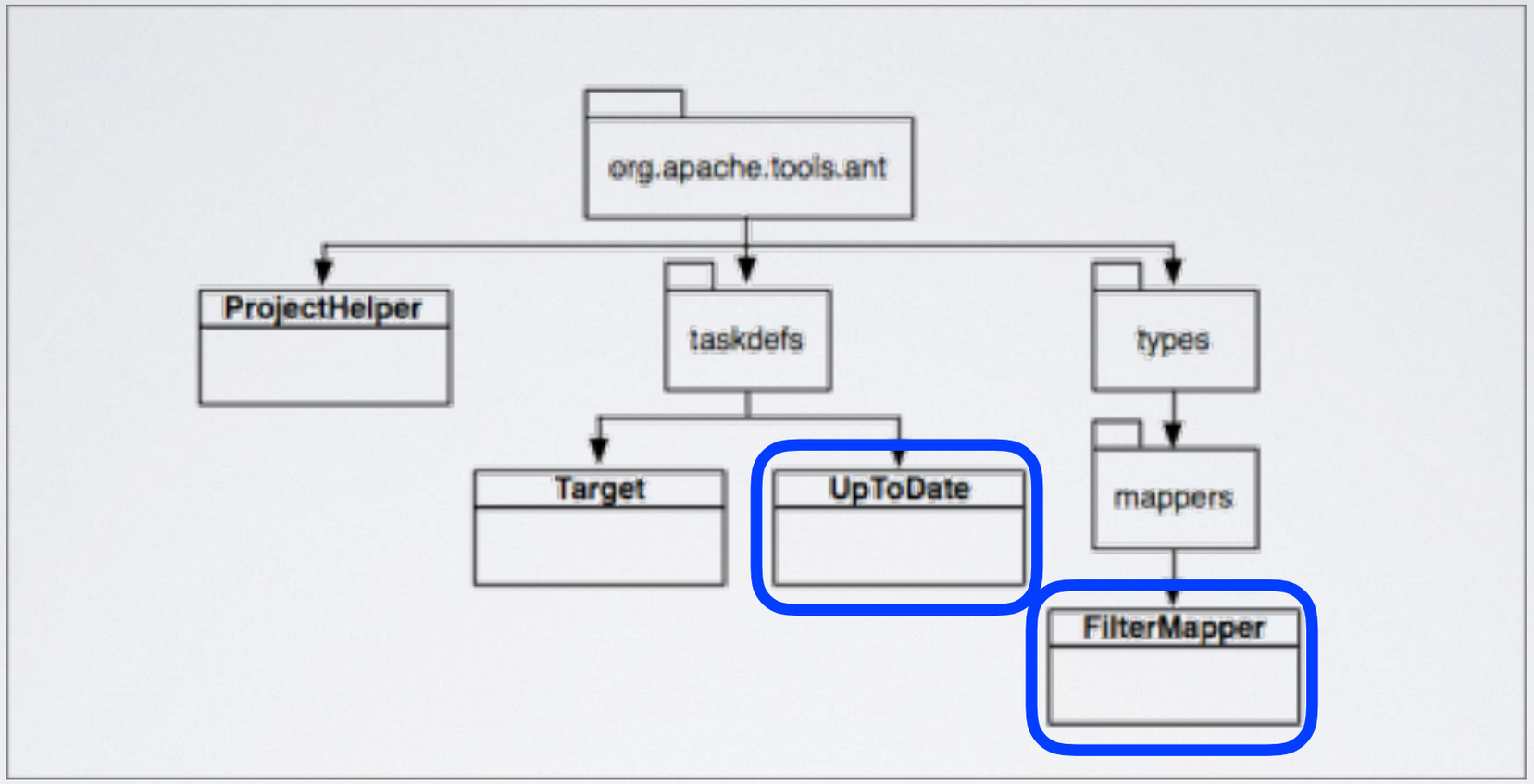
2015

Developers' Scattered Changes



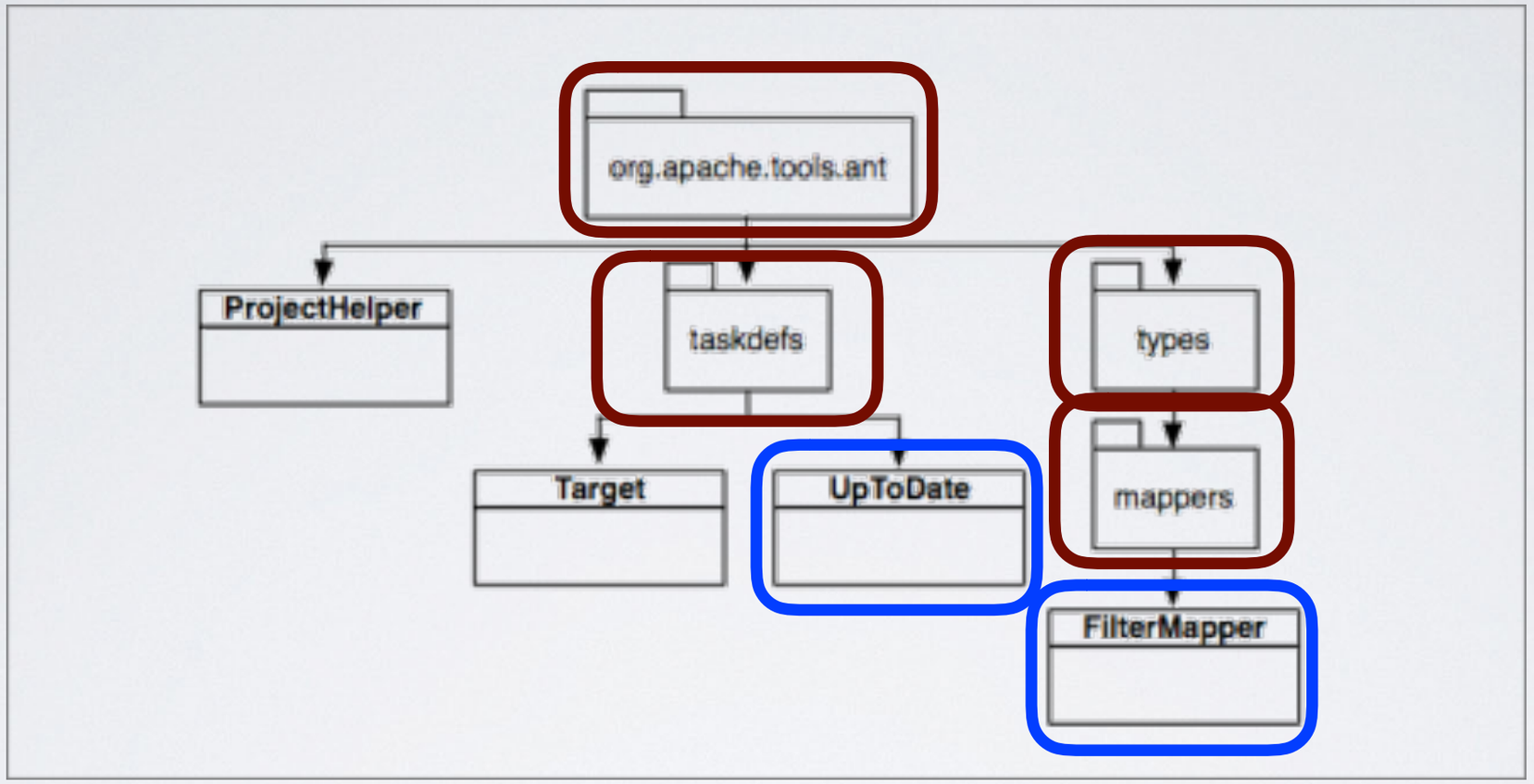
2015

Developers' Scattered Changes



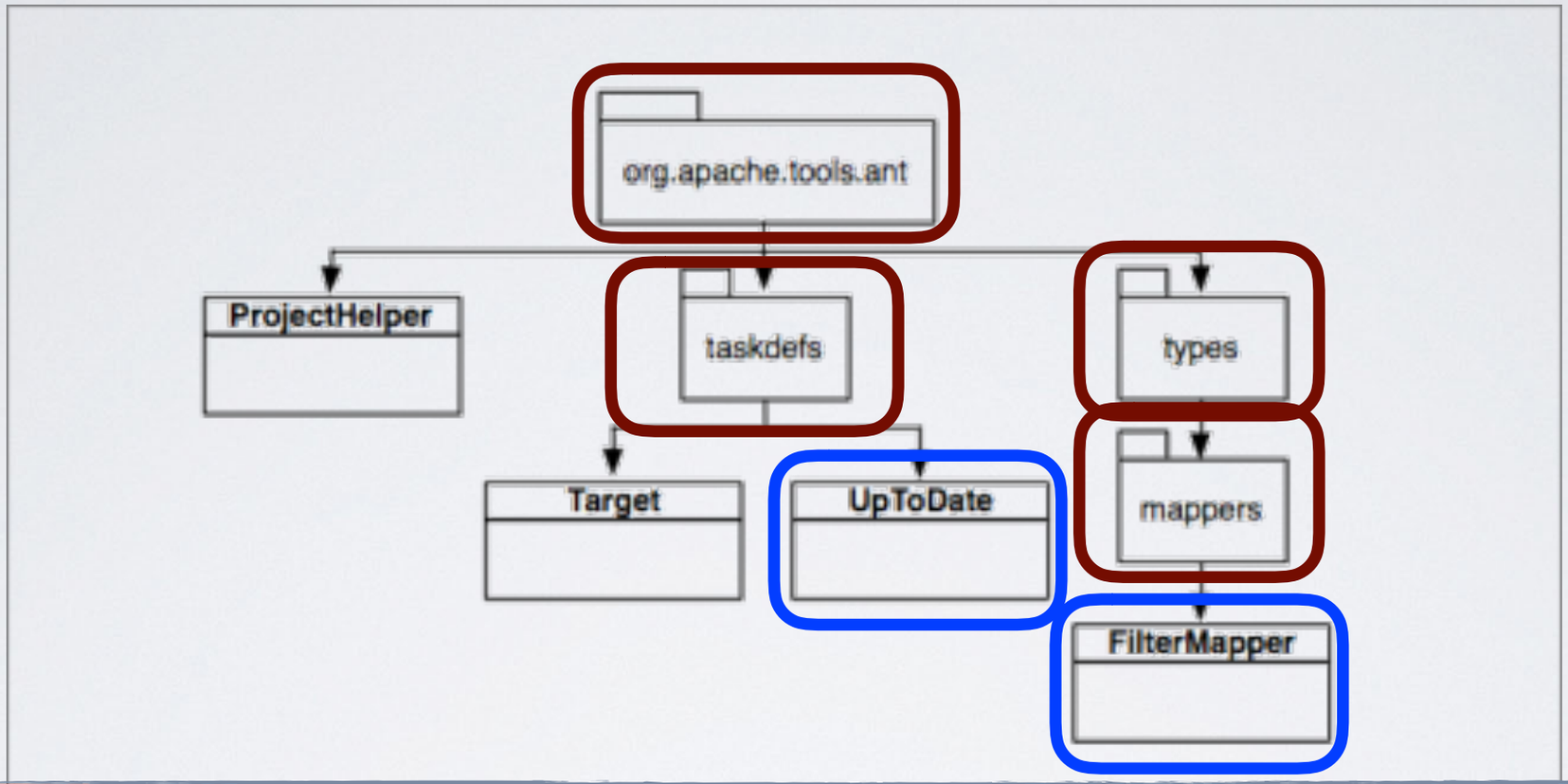
2015

Developers' Scattered Changes



2015

Developers' Scattered Changes



The structural distance is 4

2015

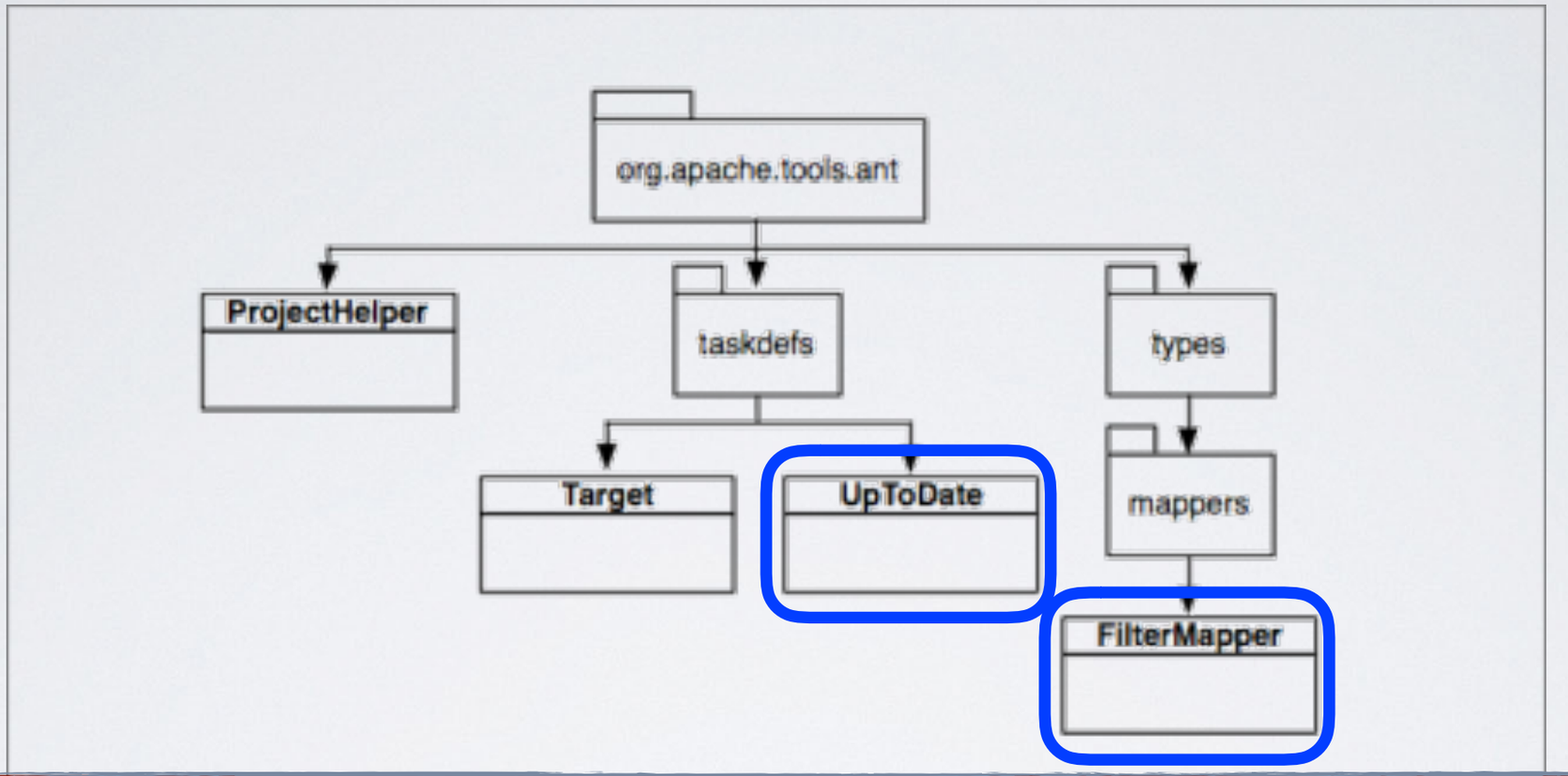
Developers' Scattered Changes

Not all the systems are well modularized…

Structural distance is not enough!

2015

Developers' Scattered Changes



We also measure the textual similarity between the artifacts modified by the developer

2015

Developers' Scattered Changes

The structural scattering of the component C in a time period p is the sum of the structural scattering of developers working on C in the period p

2015

Developers' Scattered Changes

The semantic scattering of the component C in a time period p is the sum of the semantic scattering of developers working on C in the period p

2015

Developers' Scattered Changes

A Developer Centered Bug Prediction Model

Dario Di Nucci[1], Fabio Palomba[1], Giuseppe De Rosa[1]
Gabriele Bavota[2], Rocco Oliveto[3], Andrea De Lucia[1]

[1]University of Salerno, Fisciano (SA), Italy, [2]Università della Svizzera italiana (USI), Switzerland,
[3]University of Molise, Pesche (IS), Italy

ddinucci@unisa.it, fpalomba@unisa.it, giuderos@gmail.com
gabriele.bavota@usi.ch, rocco.oliveto@unimol.it, adelucia@unisa.it

Abstract—Several techniques have been proposed to accurately predict software defects. These techniques generally exploit characteristics of the code artefacts (i.e., size, complexity, etc.) and/or of the process adopted in their development and maintenance (i.e., the number of developers working on a component) to spot out components likely containing bugs. While these bug prediction models achieve good levels of accuracy, they mostly ignore the major role played by human-related factors in the introduction of bugs. Previous studies have demonstrated that focused developers are less prone to introduce defects than non-focused developers. According to this observation, software components changed by focused developers should also be less error prone than components changed by less focused developers. We capture this observation by measuring the scattering of changes performed by developers working on a component and use this information to build a bug prediction model. Such a model has been evaluated on 26 systems and compared with four competitive techniques. The achieved results show the superiority of our model, and its high complementarity with respect to predictors commonly used in the literature. Based on this result, we also show the results of a "hybrid" prediction model combining our predictors with the baselines.

Index Terms—Scattering Metrics, Bug Prediction, Empirical Study, Mining Software Repositories

1 Introduction

Bug prediction techniques are used to identify areas of software systems that are more likely to contain bugs. These prediction models represent an important aid when the resources available for testing are scarce, since they can indicate *where* to invest such resources. The scientific community has developed several bug prediction models that can be roughly classified into two families, based on the information they exploit to discriminate between "buggy" and "clean" code components. The first set of techniques exploits *product metrics* (i.e., metrics capturing intrinsic characteristics of the code components, like their size and complexity) [1], [2], [3], [4], [5], while the second one focuses on *process metrics* (i.e., metrics capturing specific aspects of the development process, like the frequency of changes performed to code components) [6], [7], [8], [9], [10], [11], [12]. While some studies highlighted the superiority of these latter with respect to the *product metric based* techniques [7], [13], [11] there is a general consensus on the fact that no technique is the best in all contexts [14], [15]. For this reason, the research community is still spending effort in investigating under which circumstances and during which coding activities developers tend to introduce bugs (see e.g., [16], [17], [18], [19], [20], [21], [22]).

Some of these studies have highlighted the central role played by developer-related factors in the introduction of bugs. In particular, Eyolfson *et al.* [17] showed that more experienced developers tend to introduce less faults in software systems. Rahman and Devanbu [18] partly contradict the study by Eyolfson *et al.* by showing that the experience of a developer has no clear association with the introduction of buggy code. Bird *et al.* [20] found that high levels of ownership are associated with fewer bugs. Finally, Posnett *et al.* [22] showed that focused developers (i.e., developers focusing their attention on a specific part of the system) introduce fewer bugs than unfocused developers.

Although such studies showed the potential of human-related factors in bug prediction, in our previous research [23] we observed that this information is not captured in state-of-the-art bug prediction models based on process metrics extracted from version history. Indeed, previous bug prediction models have exploited predictors based (i) on the number of developers working on a code component [9] [10]; (ii) on the analysis of change-proneness [13] [11] [12]; and (iii) on the entropy of changes [8]. Thus, despite the previously discussed finding by Posnett *et al.* [22], none of the proposed bug prediction models consider how focused the developers performing changes are and how scattered these changes are. In our previous work [23] we studied the role played by *scattered changes* in bug prediction. Specifically, we firstly defined two measures, namely the developer's *structural* and *semantic scattering*. The first aims at assessing how "structurally far" the code components modified in a given time period by

Scattering metrics are orthogonal to other metrics

The prediction model built using these metrics obtain higher performance than baseline process metric-based models

2015

Research provides different alternatives

Structural metrics

Complexity metrics

Process metrics

Developer-based metrics

Textual metrics

Role of code comments

Classifying code comments in Java open-source software systems

Luca Pascarella
Delft University of Technology
Delft, The Netherlands
L.Pascarella@tudelft.nl

Alberto Bacchelli
Delft University of Technology
Delft, The Netherlands
A.Bacchelli@tudelft.nl

***Abstract*—Code comments are a key software component containing information about the underlying implementation. Several studies have shown that code comments enhance the readability of the code. Nevertheless, not all the comments have the same goal and target audience. In this paper, we investigate how six diverse Java OSS projects use code comments, with the aim of understanding their purpose. Through our analysis, we produce a taxonomy of source code comments; subsequently, we investigate how often each category occur by manually classifying more than 2,000 code comments from the aforementioned projects. In addition, we conduct an initial evaluation on how to automatically classify code comments at line level into our taxonomy using machine learning; initial results are promising and suggest that an accurate classification is within reach.**

I. Introduction

While writing and reading source code, software engineers routinely introduce code comments [6]. Several researchers investigated the usefulness of these comments, showing that thoroughly commented code is more readable and maintainable. For example, Woodfield *et al.* conducted one of the first experiments demonstrating that code comments improve program readability [35]; Tenny *et al.* confirmed these results with more experiments [31], [32]. Hartzman *et al.* investigated the economical maintenance of large software products showing that comments are crucial for maintenance [12]. Jiang *et al.* found that comments that are misaligned to the annotated functions confuse authors of future code changes [13]. Overall, given these results, having abundant comments in the source code is a recognized good practice [4]. Accordingly, researchers proposed to evaluate code quality with a new metric based on code/comment ratio [21], [9].

Nevertheless, not all the comments are the same. This is evident, for example, by glancing through the comments in a source code file[1] from the Java Apache Hadoop Framework [1]. In fact, we see that some comments target end-user programmers (*e.g.*, Javadoc), while others target internal developers (*e.g.*, *inline* comments); moreover, each comment is used for a different purpose, such as providing the implementation rationale, separating logical blocks, and adding reminders; finally, the interpretation of a comment also depends on its position with respect to the source code.

Defining a taxonomy of the source code comments that developers produce is an open research problem.

[1]https://tinyurl.com/zqeqgpq

Haouari *et al.* [11] and Steidl *et al.* [28] presented the earliest and most significant results in comments' classification. Haouari *et al.* investigated developers' commenting habits, focusing on the position of comments with respect to source code and proposing an initial taxonomy that includes four high-level categories [11]; Steidl *et al.* proposed a semi-automated approach for the quantitative and qualitative evaluation of comment quality, based on classifying comments in seven high-level categories [28]. In spite of the innovative techniques they proposed to both understanding developers' commenting habits and assessing comments' quality, the classification of comments was not in their primary focus.

In this paper, we focus on increasing our empirical understanding of the types of comments that developers write in source code files. This is a key step to guide future research on the topic. Moreover, this increased understanding has the potential to (1) improve current quality analysis approaches that are restricted to the comment ratio metric only [21], [9] and to (2) strengthen the reliability of other mining approaches that use source code comments as input (*e.g.*, [30], [23]).

To this aim, we conducted an in-depth analysis of the comments in the source code files of six major OSS systems in Java. We set up our study as an exploratory investigation. We started without hypotheses regarding the content of source code comments, with the aim of discovering their purposes and roles, their format, and their frequency. To this end, we (1) conducted three iterative content analysis sessions (involving four researchers) over 50 source files including about 250 comment blocks to define an initial taxonomy of code comments, (2) validated the taxonomy externally with 3 developers, (3) inspected 2,000 source code files and manually classified (using a new application we devised for this purpose) over 15,000 comment blocks comprising more than 28,000 lines, and (4) used the resulting dataset to evaluate how effectively comments can be automatically classified.

Our results show that developers write comments with a large variety of different meanings and that this should be taken into account by analyses and techniques that rely on code comments. The most prominent category of comments summarizes the purpose of the code, confirming the importance of research related to automatically creating this type of comments. Finally, our automated classification approach reaches promising initial results.

Code comments have different meanings

2017

Role of code comments

```
(22) // Remove sub launch, keeping the processes of the terminated launch to
(23) // show the association and to keep the console content accessible
(24) if (subLaunches.remove(launch) != null) {
(25)     // terminate ourselves if this is the last sub launch
(26)     if (subLaunches.size() == 0) {
(27)         // TODO: Check the possibility to exclude it
(28)         //monitor.exlude();
(29)         monitor.subTask("Terminated"); //$NON-NLS-1$
(30)         fTerminated = true;
(31)         fireTerminate();
(32)         // %%%
(33)         }
(34)     }
```

Summarize code

Explain experiment

Not to be read

Type declaration mismatches

Investigating Type Declaration Mismatches in Python

Luca Pascarella, Achyudh Ram
Delft University of Technology
The Netherlands
{L.Pascarella, A.R.Keshavram-1}
@tudelft.nl

Azqa Nadeem, Dinesh Bisesser, Norman Knyazev
Delft University of Technology
The Netherlands
{A.Nadeem, S.P.D.Bisesser, N.Knyazev}
@student.tudelft.nl

Alberto Bacchelli
University of Zurich
Switzerland
bacchelli@ifi.uzh.ch

***Abstract*—Past research provided evidence that developers making code changes sometimes omit to update the related documentation, thus creating inconsistencies that may contribute to faults and crashes. In dynamically typed languages, such as Python, an inconsistency in the documentation may lead to a mismatch in type declarations only visible at runtime.**

With our study, we investigate how often the documentation is inconsistent in a sample of 239 methods from five Python open-source software projects. Our results highlight that more than 20% of the comments are either partially defined or entirely missing and that almost 1% of the methods in the analyzed projects contain type inconsistencies. Based on these results, we create a tool, PyID, to early detect type mismatches in Python documentation and we evaluate its performance with our oracle.

I. Introduction

Creating a proper software system is a big challenge [3], [16]. To support, quicken, and ease development, software engineers often rely on the work of external developers who write software, such as libraries and remote services, also known as Application Programming Interfaces (API). These APIs are often provided with as an aid in understanding how to use them properly [21].

Several researchers conducted interviews, surveys, and experiments to define how much of this documentation is enough [28]. Recently, de Souza *et al.* investigated [4] the impact of the *agile* product development method on software documentation, confirming that source code and annexes code comments are the most important artifacts used by developers in maintainability processes. Forward and Lethbridge [13] conducted a survey discovering that also dated documentation may be relevant, nevertheless, referring to a not up to date documentation may be dangerous for developers. In fact, developers are found to be sometimes dangerously careless when it comes to keeping this documentation updated [13], [8]. This behavior leads to poor or *unaligned documentation*, which may create delays in the software development or, even worse, faults in software artifacts [20], [25].

The problems created by unaligned documentation become even more significant both (1) for dynamically typed languages, such as Python, where code comments provide valuable information regarding method specification for both internal and external developers [29], and (2) for API documentation where the source code is not available (*e.g.*, web APIs). Shi *et al.* explored the co-evolution of the API and related documentation of big libraries finding that the code of two nearby releases may evolve dramatically, thus requiring also a crucial evolution of the annexed documentation, which underlines the relevance of the topic.

Zhou *et al.* proposed one of the most recent investigations on the frequent inconsistencies between source code and API documentation [31]. They proposed an automated approach based on program comprehension and natural language processing to address the inconsistencies in method's parameters by creating constraints and exception throwing declarations. Such solution becomes particularly useful when integrated into IDEs, as it creates timely alerts asking developers to handle the mismatches between types declared in the documentation and types referred in the source code.

Despite the innovative technique proposed by Zhou *et al.*, their model is limited to statically typed languages, such as Java. Nevertheless, developing code in dynamically typed languages makes the code even more prone to hidden vulnerabilities stemming from code-comment inconsistencies [26]. In fact, a type mismatch may trigger an error far away from where the type mismatch initially occurred or, even worse, it may never trigger a runtime error while failing silently with serious consequences.

In the work we present in this paper, we conducted a first step in investigating and supporting the alignment between documentation and source code in dynamically typed languages. In particular, we investigated the alignment between methods and comments in five popular OSS Python projects. We started with an empirical analysis of how careful open-source software (OSS) projects developers are about maintaining aligned documentations. For this purpose, we manually inspected the alignment between methods' body and *docstring* of 239 methods from the aforementioned OSS Python projects.

Our results show that the Python developers of the five OSS systems we sampled do care about documentation. In fact, even though developers left incomplete or totally pending more than 20% of the analyzed public methods, less than 1% of the analyzed methods contains mismatches between declared and used types. This finding empirically underlines how important documentation is deemed to be by developers of dynamically typed languages. Since even a 1% of unaligned methods may become problematic, we designed PyID, an OSS tool based on machine learning aimed at helping developers to early detect type mismatches in documentation.

Code comments are ancillary to other metrics

2018

Method-level bug prediction

On the Performance of Method-Level Bug Prediction: A Negative Result

Luca Pascarella, Fabio Palomba, Alberto Bacchelli

the date of receipt and acceptance should be inserted later

Abstract Bug prediction is aimed at identifying software artifacts that are more likely to be defective. Most approaches defined so far target the prediction of bugs at class/file level. Nevertheless, past research has provided evidence that this granularity might be too coarse-grained, thus reducing the usability of bug prediction in practice. As a consequence, researchers have started proposing defect prediction models targeting a finer granularity, particularly targeting methods, providing promising evidence that it is possible to operate at this granularity. Particularly, models based on a mixture of product and process metrics provided the best results.

In this paper, we first replicate previous research on method-level bug-prediction using different systems and timespans. Afterward, based on the limitations of existing research, we (1) re-evaluate method-level bug prediction models more realistically and (2) analyze the whether textual features—previously shown as valuable sources of information for the evaluation of software quality (yet surprisingly unexplored in this research field)—can be exploited to improve method-level bug prediction abilities. Key results of our study include that (1) the performance of the previously proposed models, tested using the same strategy but with different systems/timespans, is confirmed. However, (2) when evaluated with a more realistic strategy all the models show a dramatic drop in performance exhibiting results close to that of a random classifier. In addition, we find that (3) the contribution of textual features within such models is limited and unable to improve the prediction capabilities significantly. As a consequence, our replication and negative results indicate that method-level bug prediction is still an open challenge.

Luca Pascarella
Delft University of Technology, The Netherlands
E-mail: L.Pascarella@tudelft.nl

Fabio Palomba, Alberto Bacchelli
University of Zurich, Switzerland
E-mail: palomba@ifi.uzh.ch, bacchelli@ifi.uzh.ch

Textual features boost the performance of 4% (F-Measure)

In progress…

Alternative ways…

Empir Software Eng (2012) 17:243–275
DOI 10.1007/s10664-011-9171-y

An exploratory study of the impact of antipatterns on class change- and fault-proneness

Foutse Khomh · Massimiliano Di Penta · Yann-Gaël Guéhéneuc · Giuliano Antoniol

Published online: 6 August 2011

Editor: Jim Whitehead

Abstract Antipatterns are poor design choices that are conjectured to make object-oriented systems harder to maintain. We investigate the impact of antipatterns on classes in object-oriented systems by studying the relation between the presence of antipatterns and the change- and fault-proneness of the classes. We detect 13 antipatterns in 54 releases of ArgoUML, Eclipse, Mylyn, and Rhino, and analyse (1) to what extent classes participating in antipatterns have higher odds to change or to be subject to fault-fixing than other classes, (2) to what extent these odds (if higher) are due to the sizes of the classes or to the presence of antipatterns, and (3) what kinds of changes affect classes participating in antipatterns. We show that, in almost all releases of the four systems, classes participating in antipatterns are more change- and fault-prone than others. We also show that size alone cannot explain the higher odds of classes with antipatterns to underwent a (fault-fixing) change than other

We thank Marc Eaddy for making his data on faults freely available. This work has been partly funded by the NSERC Research Chairs in Software Change and Evolution and in Software Patterns and Patterns of Software.

F. Khomh (✉)
Department of Electrical and Computer Engineering,
Queen's University, Kingston, ON, Canada
e-mail: foutse.khomh@queensu.ca

M. D. Penta
Department of Engineering, University of Sannio, Benevento, Italy
e-mail: dipenta@unisannio.it

Y.-G. Guéhéneuc · G. Antoniol
SOCCER Lab. and Ptidej Team, Département de Génie Informatique et Génie Logiciel,
École Polytechnique de Montréal, Montréal, QC, Canada

Y.-G. Guéhéneuc
e-mail: yann-gael.gueheneuc@polymtl.ca

G. Antoniol
e-mail: antoniol@ieee.org

Bad Smells increase change- and bug-proneness

[Khomh et al. - EMSE 2012]

Smells like Teen Spirit: Improving Bug Prediction Performance using the Intensity of Code Smells

Smells like Teen Spirit: Improving Bug Prediction Performance using the Intensity of Code Smells

Fabio Palomba*, Marco Zanoni†, Francesca Arcelli Fontana†, Andrea De Lucia*, Rocco Oliveto‡
*University of Salerno, Italy, †University of Milano-Bicocca, Italy, ‡University of Molise, Italy
fpalomba@unisa.it, marco.zanoni@disco.unimib.it, arcelli@disco.unimib.it, adelucia@unisa.it, rocco.oliveto@unimol.it

***Abstract*—Code smells are symptoms of poor design and implementation choices. Previous studies empirically assessed the impact of smells on code quality and clearly indicate their negative impact on maintainability, including a higher bug-proneness of components affected by code smells. In this paper we capture previous findings on bug-proneness to build a specialized bug prediction model for smelly classes. Specifically, we evaluate the contribution of a measure of the severity of code smells (*i.e.*, code smell intensity) by adding it to existing bug prediction models and comparing the results of the new model against the baseline model. Results indicate that the accuracy of a bug prediction model increases by adding the code smell intensity as predictor. We also evaluate the actual gain provided by the intensity index with respect to the other metrics in the model, including the ones used to compute the code smell intensity. We observe that the intensity index is much more important as compared to other metrics used for predicting the buggyness of smelly classes.**

I. Introduction

In the last decade, the research community has spent a lot of effort in investigating bad code smells (shortly "code smells" or simply "smells"), *i.e.*, symptoms of poor design and implementation choices applied by programmers during the development of a software project [1]. Besides approaches for the automatic identification of code smells in source code [2]–[7], empirical studies have been conducted to understand when and why code smells appear [8], the relevance they have for developers [9], [10], their evolution and longevity in software projects [11]–[14], as well as the negative effects of code smells on software understandability [15], and maintainability [16]–[19]. Recently, Khomh *et al.* [20] have also empirically demonstrated that classes affected by design problems ("antipatterns") are more prone to contain bugs in the future. Although this study showed the potential importance of code smells in the context of bug prediction, these observations have not been captured in bug prediction models yet. Indeed, while previous work has proposed the use of predictors based on product metrics (e.g., see [21]–[23]), as well as the analysis of change-proneness [24]–[26], the entropy of changes [27], or human-related factors [28]–[30] to build accurate bug prediction models, none of them takes into account a measure able to quantify the presence and the severity of design problems affecting code components.

In this paper, we aim at making a further step ahead by studying the role played by bad code smells in bug prediction. Our hypothesis is that *taking into account the severity of a design problem affecting a source code element in a bug prediction model can contribute to the correct classification of the buggyness of such a component*. To verify this conjecture, we use the intensity index (*i.e.*, a metric able to estimate the severity of a code smell) defined by Arcelli Fontana *et al.* [31] to build a bug prediction model that takes into account the presence and the severity of design problems affecting a code component. Specifically, we evaluate the predictive power of the intensity index by adding it in a bug prediction model based on structural quality metrics [32], and comparing its accuracy against the one achieved by the baseline model on six large Java open source systems. We also quantified the gain provided by the addition of the intensity index with respect to the other structural metrics in the model, including the ones used to compute the intensity. Finally, we report further analyses aimed at understanding (i) the accuracy of a model where a simple truth value reporting the presence/absence of code smells rather than the intensity index is added to the baseline model, (ii) the impact of false positive smell instances identified by the code smell detector, and (iii) the contribution of the intensity index in bug prediction models based on process metrics.

The results of our study indicate that:

- The addition of the intensity index as predictor of buggy components positively impact the accuracy of a bug prediction model based on structural quality metrics. We observed an improvement of the accuracy of the classification up to 25% as compared to the accuracy achieved by the baseline model.
- The intensity index is more important than other quality metrics for the prediction of the bug-proneness of smelly classes.
- The presence of a limited number of false positive smell instances identified by the code smell detector does not impact the accuracy and the practical applicability of the proposed specialized bug prediction model.
- The intensity index positively impacts the performance of bug prediction models based on process metrics, increasing the accuracy of the classification up to 47%.

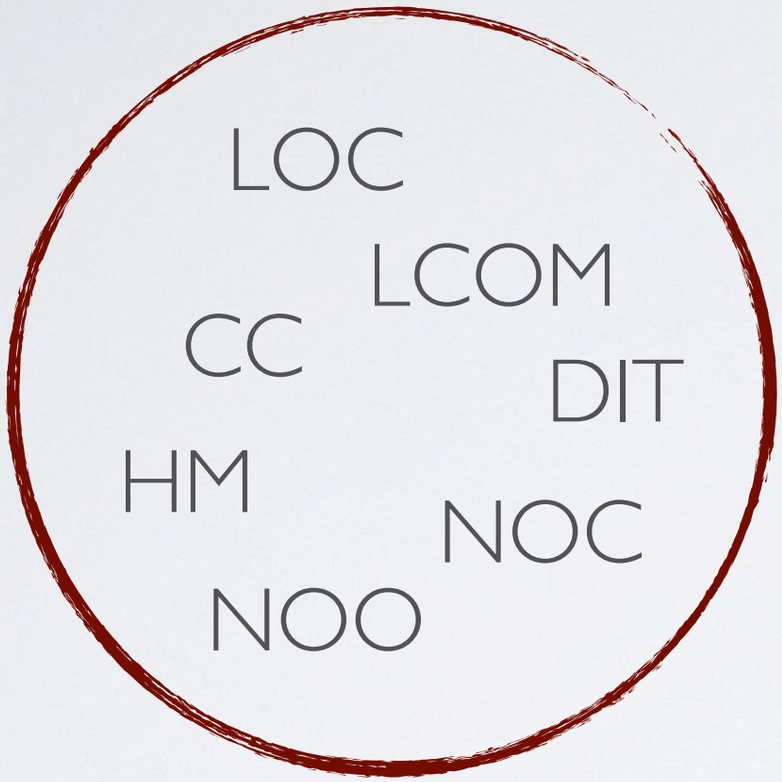
Structure of the paper. Section II discusses the related literature on bug prediction models, while Section III presents the specialized bug prediction model for *smelly* classes. Section IV describes the design and the results of the case study aimed at evaluating the accuracy of the proposed model. Section V discusses the results of the additional analyses we conducted.

The use of code smell information as additional predictor

Improving the description of code components using the severity of code smells affecting such components

2016

Smells like Teen Spirit: Improving Bug Prediction Performance using the Intensity of Code Smells



Predictors

+

0.86

Severity of the design flaw

2016

Smells like Teen Spirit: Improving Bug Prediction Performance using the Intensity of Code Smells

+25%

on average in terms of accuracy of the predictions

2016

Fine-grained just-in-time defect prediction

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Fine-grained just-in-time defect prediction

Luca Pascarella[a], Fabio Palomba[b,*], Alberto Bacchelli[b]

[a] Delft University of Technology, The Netherlands
[b] University of Zurich, Switzerland

ARTICLE INFO

Article history:
Received 11 May 2018
Revised 13 October 2018
Accepted 2 December 2018
Available online 3 December 2018

Keywords:
Just-in-time defect prediction
Empirical Software Engineering
Mining software repositories

ABSTRACT

Defect prediction models focus on identifying defect-prone code elements, for example to allow practitioners to allocate testing resources on specific subsystems and to provide assistance during code reviews. While the research community has been highly active in proposing metrics and methods to predict defects on *long-term* periods (*i.e.*,at release time), a recent trend is represented by the so-called *short-term* defect prediction (*i.e.*,at commit-level). Indeed, this strategy represents an effective alternative in terms of effort required to inspect files likely affected by defects. Nevertheless, the granularity considered by such models might be still too coarse. Indeed, existing commit-level models highlight an *entire* commit as defective even in cases where only *specific* files actually contain defects.

In this paper, we first investigate to what extent commits are partially defective; then, we propose a novel *fine-grained just-in-time* defect prediction model to predict the specific files, contained in a commit, that are defective. Finally, we evaluate our model in terms of (i) performance and (ii) the extent to which it decreases the effort required to diagnose a defect. Our study highlights that: (1) defective commits are frequently composed of a mixture of defective and non-defective files, (2) our fine-grained model can accurately predict defective files with an AUC-ROC up to 82% and (3) our model would allow practitioners to save inspection efforts with respect to standard just-in-time techniques.

1. Introduction

During software maintenance and evolution, developers constantly modify the source code to introduce new features or fix defects (Lehman, 1980). These modifications, however, may lead to the introduction of new defects (Kim et al., 2008), thus developers must carefully verify that the performed modifications do not introduce new defects in the code. This task is usually performed directly during development (*e.g.*,by running test cases) (Sommerville, 2006) or when changes are reviewed (Bacchelli and Bird, 2013). An efficient way to allocate inspection and testing resources to the portion of source code more likely to be defective is represented by *defect prediction* (Hall et al., 2012), which involves the construction of statistical models to predict the defect-proneness of software artifacts, by mostly exploiting information regarding the source code or the development process.

The problem of defect prediction has attracted the attention of many researchers in the past decade, who tried to address it by (i) conducting empirical studies on the factors making artifacts more defect-prone (*e.g.*, Basili et al., 1996; Khomh et al., 2012; Palomba et al., 2018; Palomba et al., 2017; Posnett et al., 2013; Spadini et al., 2018b; Tufano et al., 2017) and (ii) proposing novel prediction models aimed at accurately predicting the defect-proneness of the source code (*e.g.*, Bell et al., 2013; Hassan, 2009; Menzies et al., 2013; di Nucci et al., 2017; Palomba et al., 2016; Rahman and Devanbu, 2013).

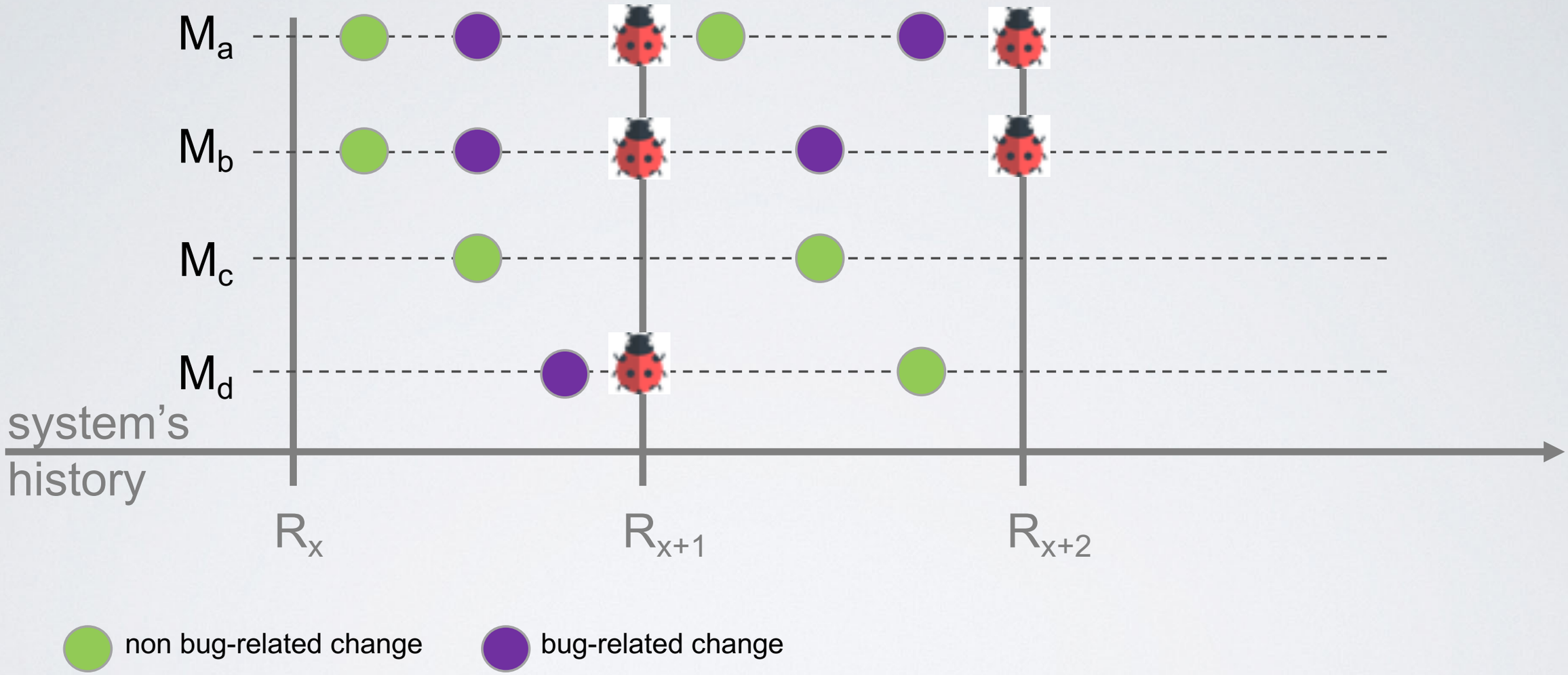
Most of the existing techniques evaluate the defectiveness of software artifacts perform *long-term* predictions. Analyzing the information accumulated in previous software releases, these models predict which artifacts are going be more prone to defect *in future releases*. For instance, Basili et al. investigated the effectiveness of Object-Oriented metrics (Chidamber and Kemerer, 1994) in predicting post-release defects (Basili et al., 1996), while other approaches consider process metrics (*e.g.*,the entropy of changes Hassan, 2009) or developer-related factors (Bell et al., 2013; di Nucci et al., 2017) for the same purpose.

Kamei et al. reported that these long-term defect prediction models—despite their good accuracy—may have a limited use-

Just-in-Time is more useful for real environments

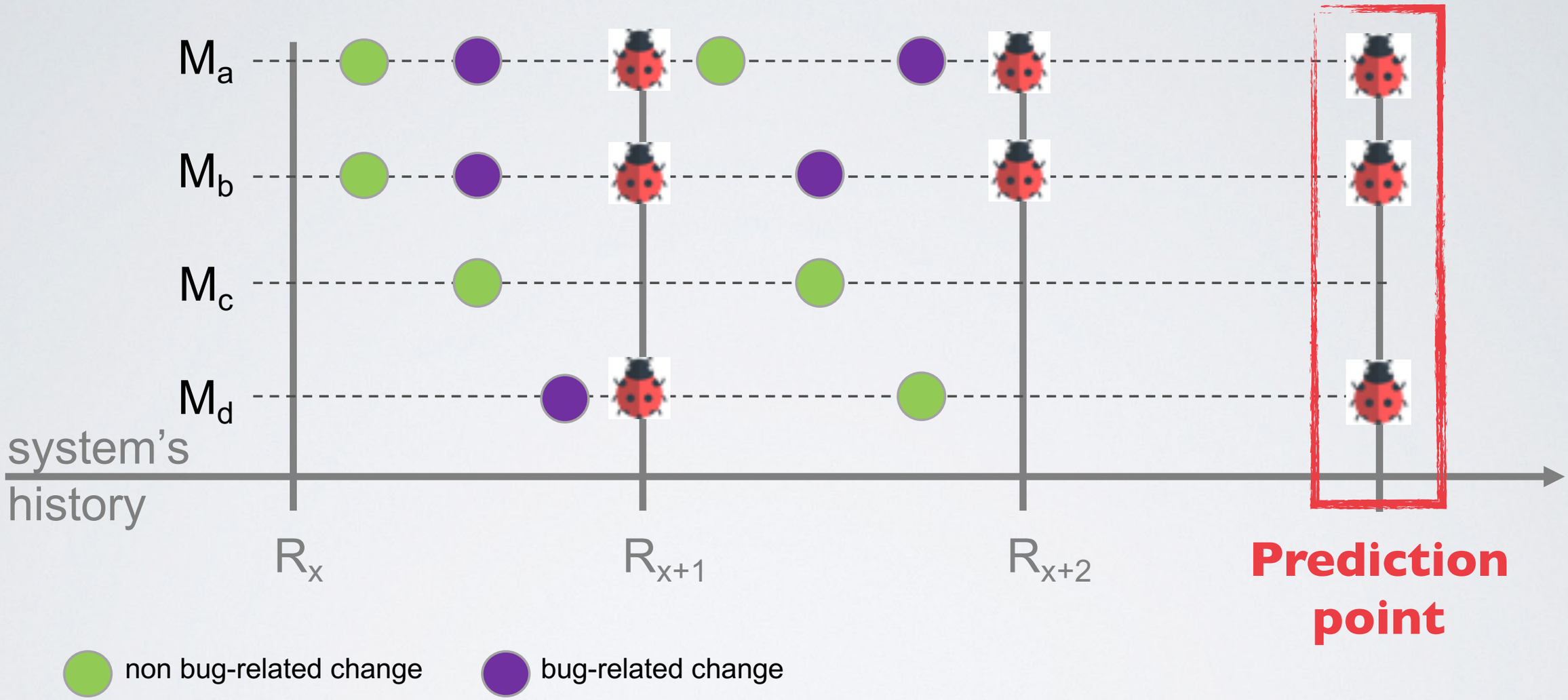
2018

Fine-grained just-in-time defect prediction



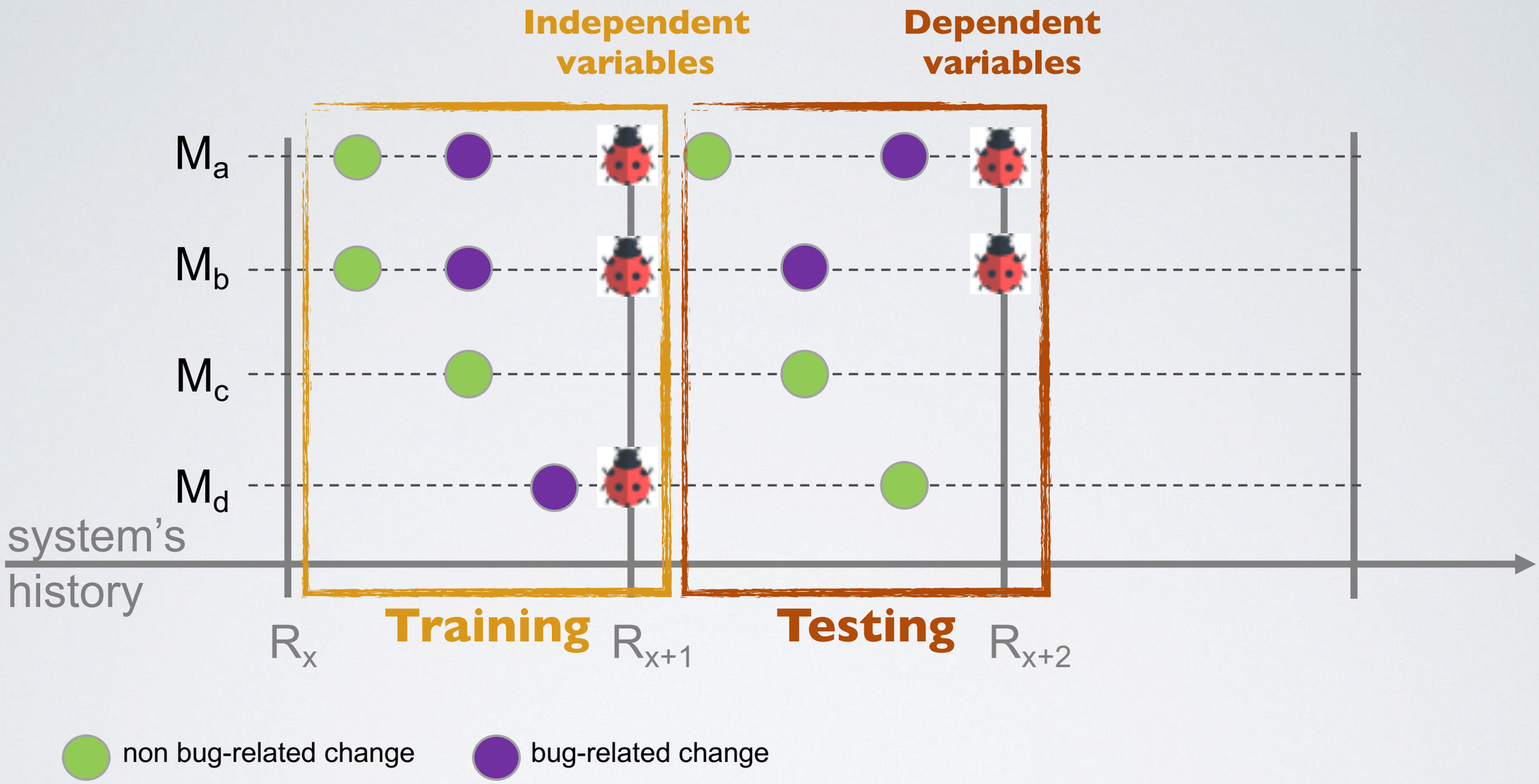
2018

Fine-grained just-in-time defect prediction



2018

Fine-grained just-in-time defect prediction



2018

Summarizing…

Everyone claim that their approach
is more efficient than others!

	Structural Metrics	Complexity Metrics	Process Metrics	Developer-based Metrics
Structural Metrics				
Complexity Metrics				
Process Metrics				
Developer-based Metrics				

Summarizing…

The silver bullet does not exist!

Summarizing…

The silver bullet does not exist!

Different predictors are suitable in different contexts

Summarizing…

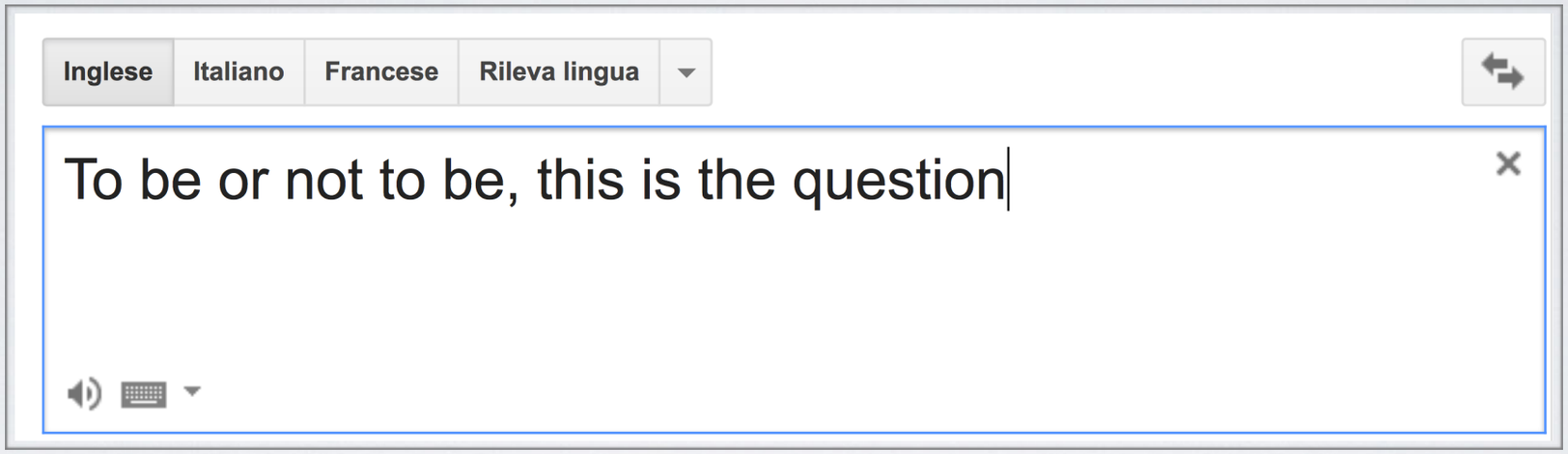
The silver bullet does not exist!

Different predictors are suitable in different contexts

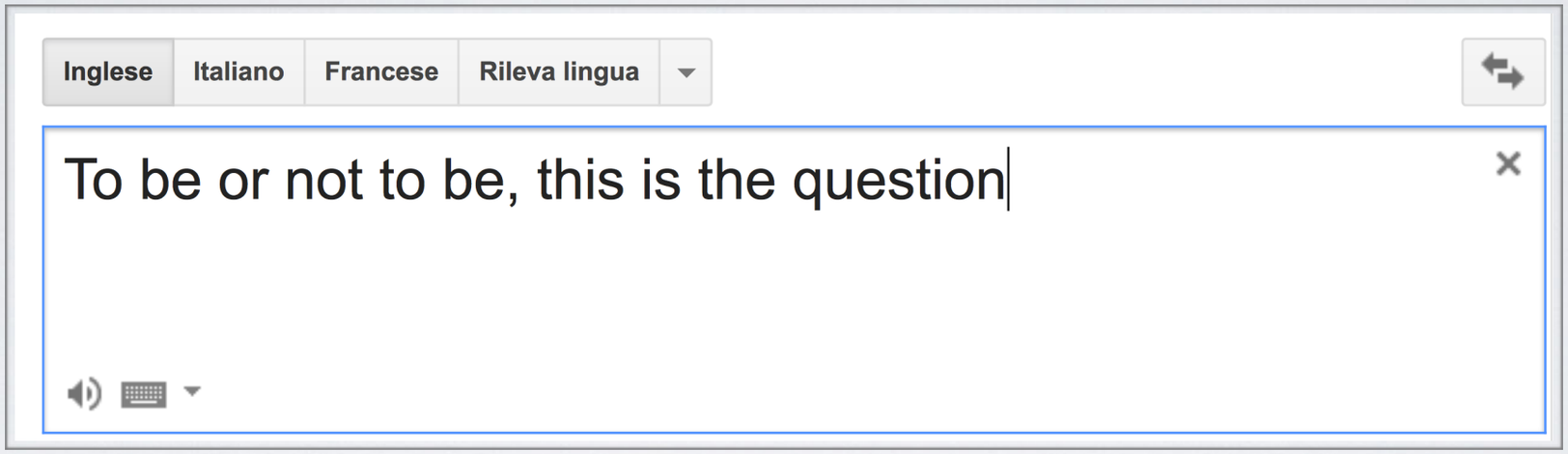
A technique which combine predictors could be very worthwhile

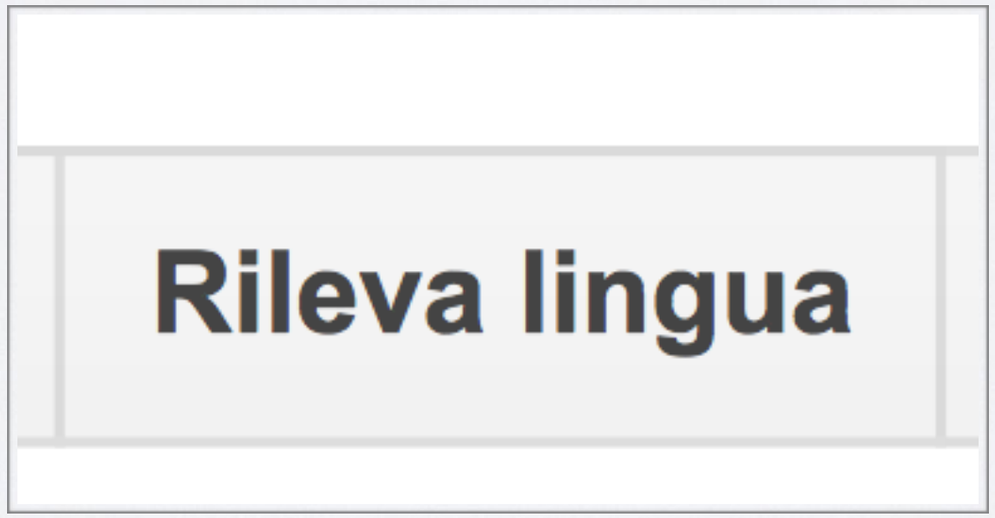
The choice of the machine learner to use

What is a machine learner?



What is a machine learner?





What is a machine learner?

The quick brown fox jumped over
the lazy dog.

What is a machine learner?

The quick brown fox jumped over the lazy dog.

What is a machine learner?

The quick brown fox jumped over the lazy dog.

To err is human, but to really foul things up you need a computer.

What is a machine learner?

The quick brown fox jumped over the lazy dog.

To err is human, but to really foul things up you need a computer.

English

English

What is a machine learner?

The quick brown fox jumped over the lazy dog.

To err is human, but to really foul things up you need a computer.

No hay mal que por bien no venga.

English

English

What is a machine learner?

The quick brown fox jumped over the lazy dog.

English

To err is human, but to really foul things up you need a computer.

English

No hay mal que por bien no venga.

Spanish

What is a machine learner?

The quick brown fox jumped over the lazy dog.

English

To err is human, but to really foul things up you need a computer.

English

No hay mal que por bien no venga.

Spanish

La tercera es la vencida.

What is a machine learner?

The quick brown fox jumped over the lazy dog.

English

To err is human, but to really foul things up you need a computer.

English

No hay mal que por bien no venga.

Spanish

La tercera es la vencida.

Spanish

What is a machine learner?

The quick brown fox jumped over the lazy dog.

English

To err is human, but to really foul things up you need a computer.

English

No hay mal que por bien no venga.

Spanish

La tercera es la vencida.

Spanish

To be or not to be, this is the question

?

What is a machine learner?

The quick brown fox jumped over the lazy dog.

English

To err is human, but to really foul things up you need a computer.

English

No hay mal que por bien no venga.

Spanish

La tercera es la vencida.

Spanish

To be or not to be, this is the question

?

La fe mueve montanas

?

What is a machine learner?

The quick brown fox jumped over the lazy dog.

English

To err is human, but to really foul

English

Training set

NO hay mal que por bien no venga.

Spanish

La tercera es la vencida.

Spanish

To be or not to be, this is the question

?

La fe mueve montanas

?

What is a machine learner?

The quick brown fox jumped over the lazy dog.

English

To err is human, but to really foul

English

Training set

¡NO hay mal que por bien no venga.

Spanish

La tercera es la vencida.

Spanish

Model f(x)

To be or not to be, this is the question

?

La fe mueve montanas

?

What is a machine learner?

The quick brown fox jumped over the lazy dog.

To err is human, but to really foul th

NO hay mal que por bien no venga.

La tercera es la vencida.

English

English

Spanish

Spanish

Training set

Model f(x)

To be or not to be, this is

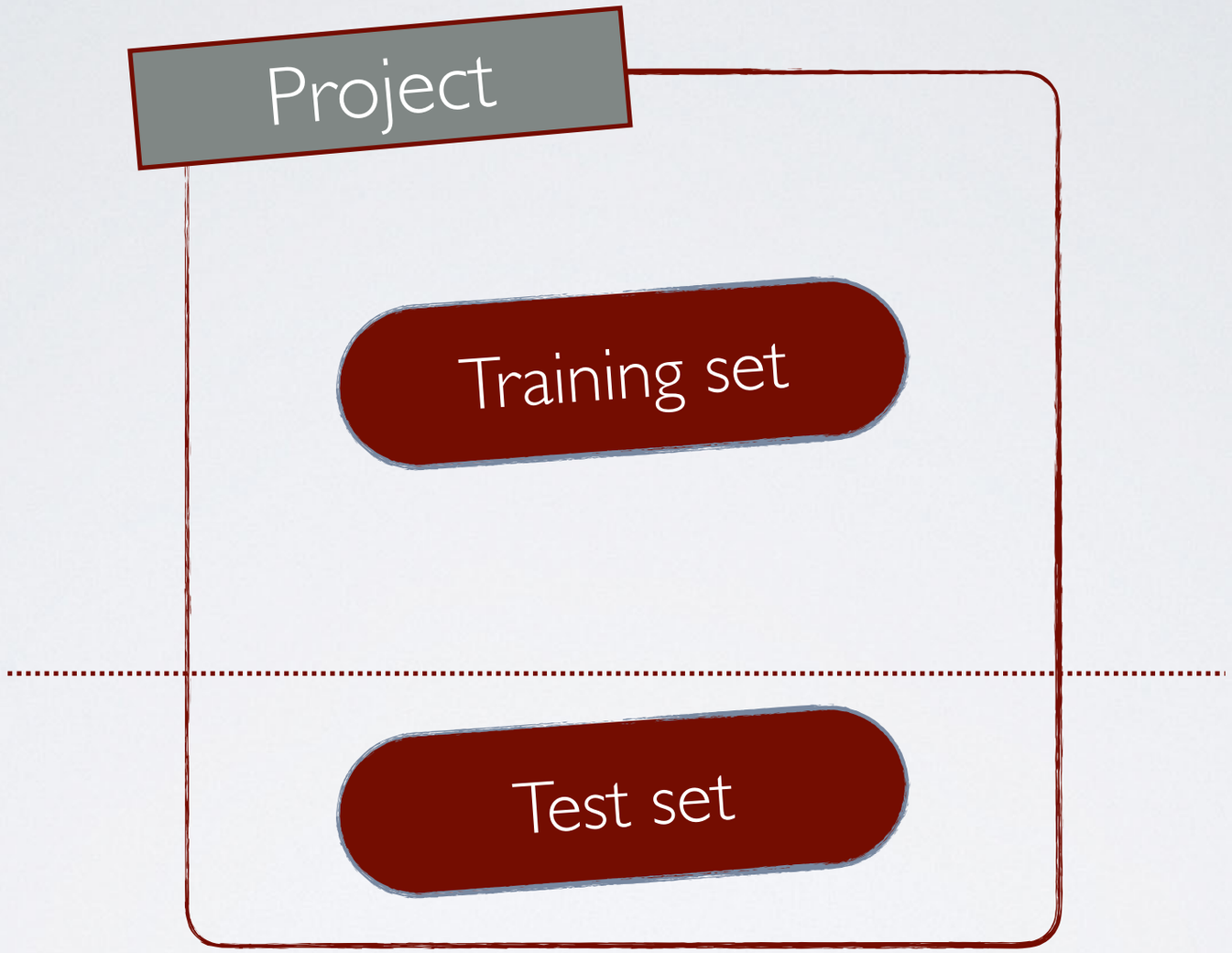
La fe mueve montanas

English?

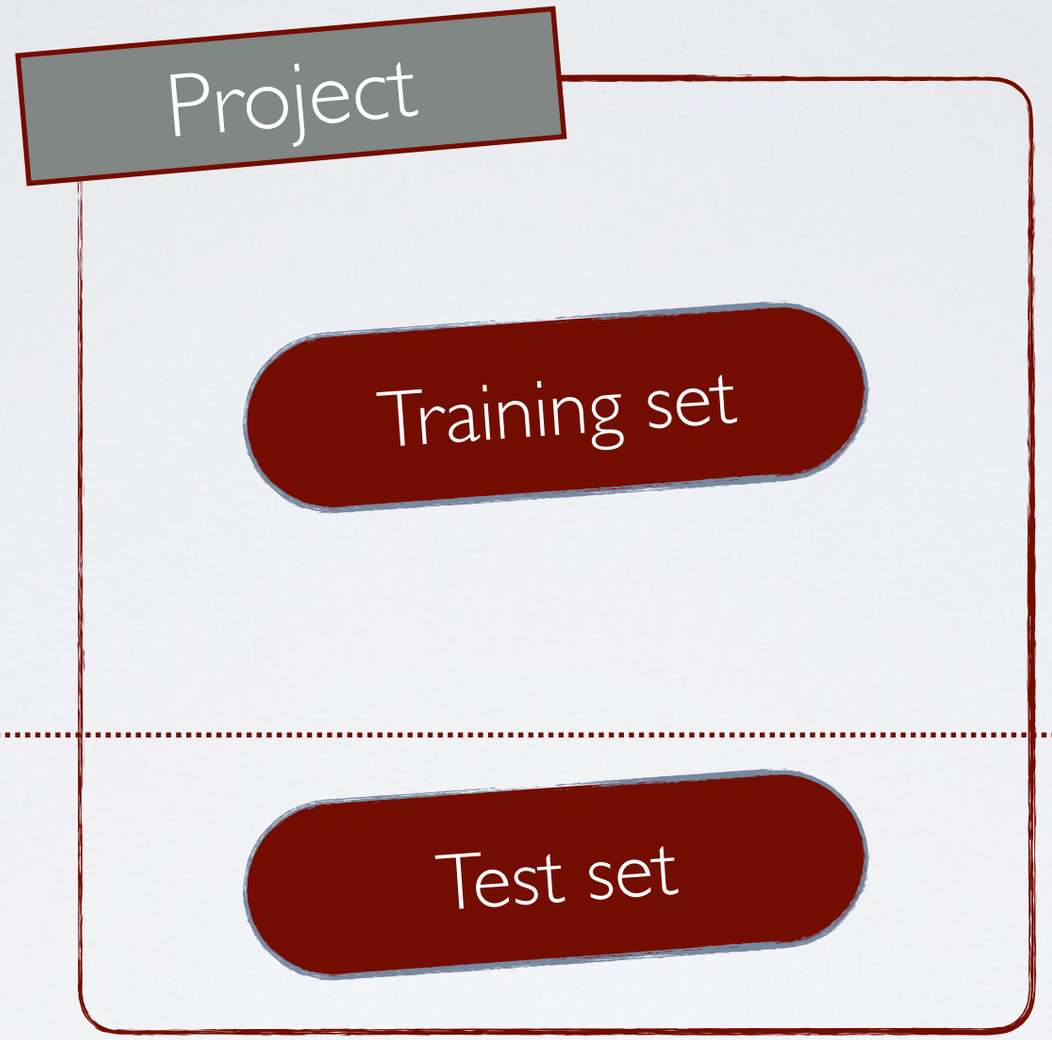
Spanish?

Test set

How to train a machine learner?

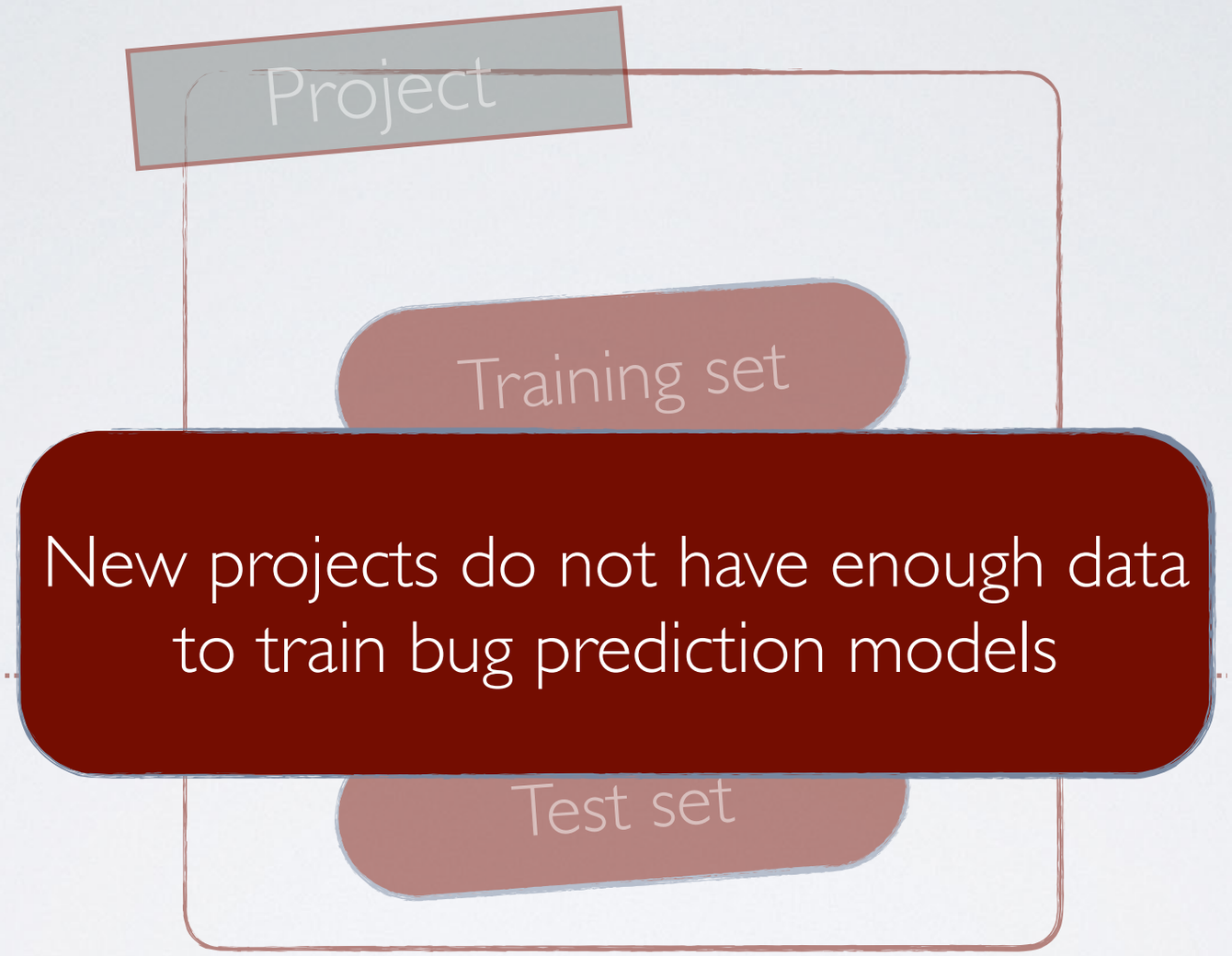


How to train a machine learner?



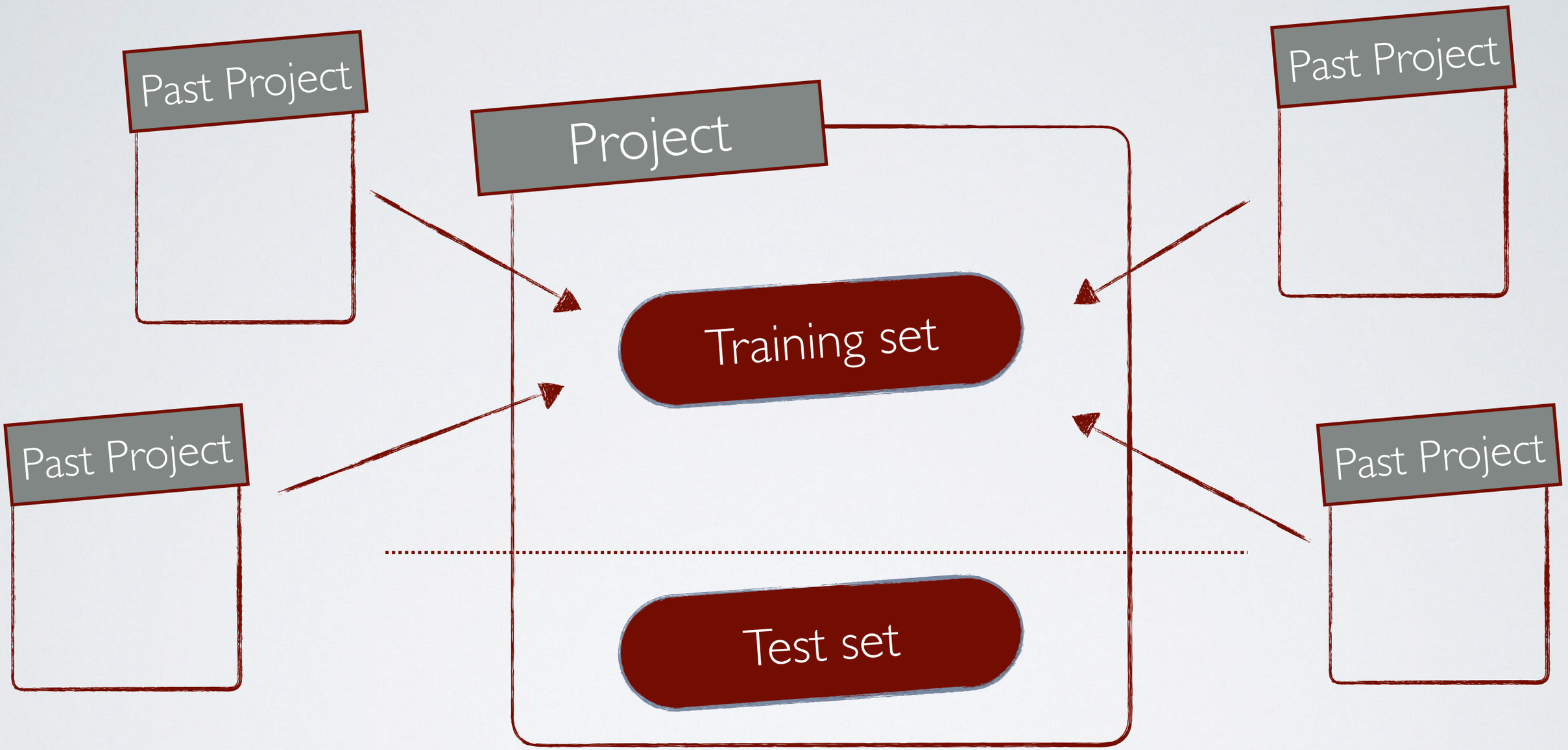
Within project

How to train a machine learner?

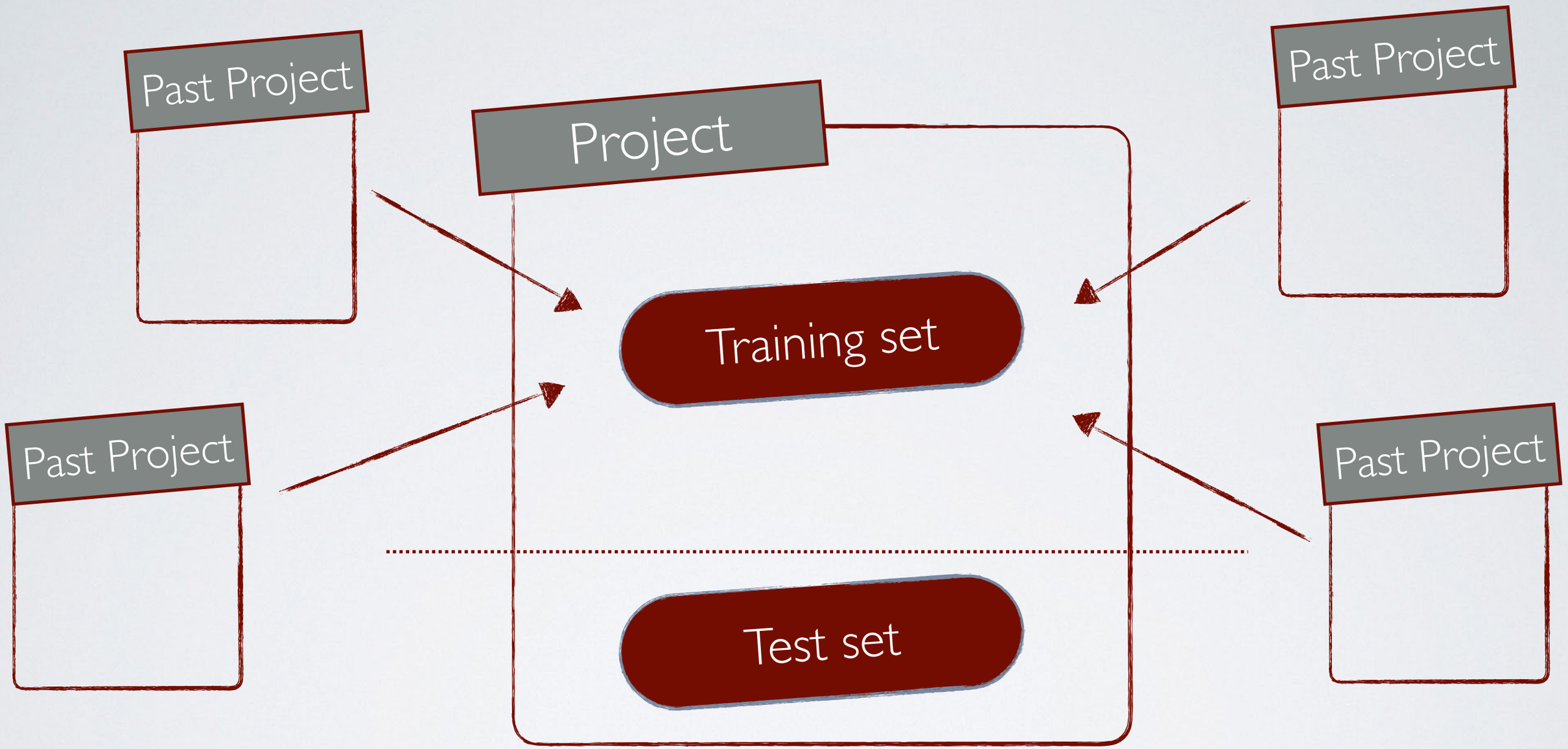


Within project

How to train a machine learner?

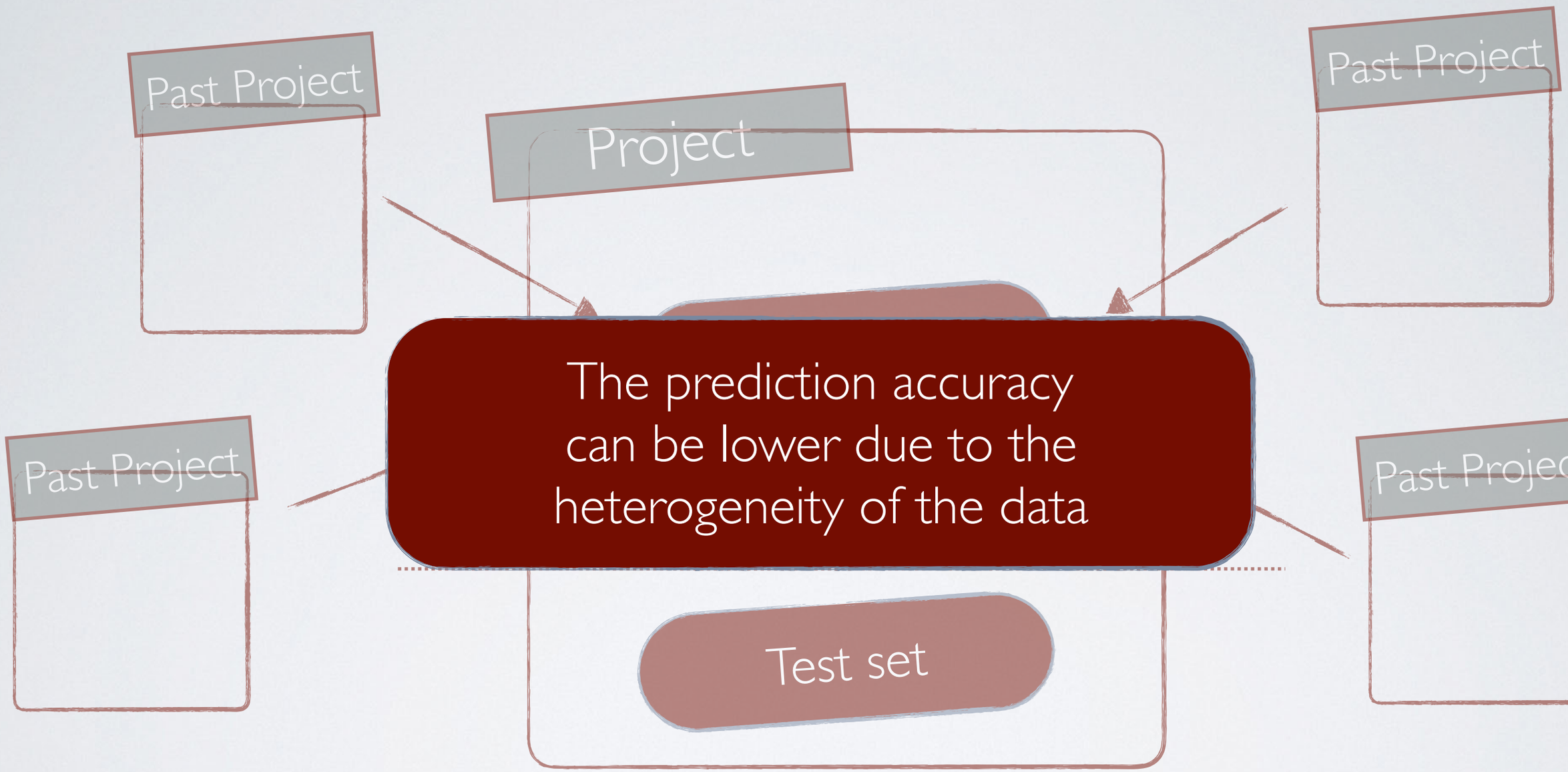


How to train a machine learner?



Cross-project

How to train a machine learner?



Cross-project

The impact of Classifiers

Cross-Project Defect Prediction Models: L'Union Fait la Force

Annibale Panichella[1], Rocco Oliveto[2], Andrea De Lucia[1]
[1]Department of Management & Information Technology, University of Salerno, Fisciano (SA), Italy
[2]Department of Bioscience and Territory, University of Molise, Pesche (IS), Italy
apanichella@unisa.it, rocco.oliveto@unimol.it, adelucia@unisa.it

***Abstract*—Existing defect prediction models use product or process metrics and machine learning methods to identify defect-prone source code entities. Different classifiers (*e.g.,* linear regression, logistic regression, or classification trees) have been investigated in the last decade. The results achieved so far are sometimes contrasting and do not show a clear winner. In this paper we present an empirical study aiming at statistically analyzing the equivalence of different defect predictors. We also propose a combined approach, coined as CODEP (COmbined DEfect Predictor), that employs the classification provided by different machine learning techniques to improve the detection of defect-prone entities. The study was conducted on 10 open source software systems and in the context of cross-project defect prediction, that represents one of the main challenges in the defect prediction field. The statistical analysis of the results indicates that the investigated classifiers are not equivalent and they can complement each other. This is also confirmed by the superior prediction accuracy achieved by CODEP when compared to stand-alone defect predictors.**

I. Introduction

During software development and maintenance, software engineers need to manage time and resources. Improper management of these factors could lead to project failure. Given its importance, in recent years a lot of effort has been devoted to provide software engineers with tools supporting management activities. Defect prediction is one of these tools. Knowing the parts of a software system that are more defect-prone may aid in scheduling testing or refactoring activities. In fact, it is reasonable to allocate more resources on more critical and defect-prone source code entities.

Existing prediction models use different information (*i.e.,* source code metrics, process metrics, or past defects) and machine learning methods to identify defect-prone source code entities. A plethora of classifiers (*e.g.,* linear regression, logistic regression, or classification trees) have been proposed and empirically investigated in the last decade [1], [2], [3]. Generally, the evaluation of these classifiers is based only on metrics—such as precision and recall—able to capture the prediction accuracy of the experimented approaches. The results achieved are sometimes contrasting and generally do not allow to identify a method that sensibly overcomes the others (see *e.g.,* [1]). Indeed, looking at the prediction accuracy (captured by metrics), the impression is that all the classifiers exploited so far are able to capture the same information when used to detect software entities having a high likelihood to exhibit faults. Thus, selecting a specific classifier does not seem to play an important role [4], [1].

However, metrics do not tell the whole story. They only provide an overview of the prediction accuracy without emphasizing peculiarities (if any) of each investigated method. What is missing is a thorough analysis of the predictions of each approach aiming at verifying whether or not different classifiers are able to identify different defect-prone entities. In other words, while different classifiers achieve the same accuracy, the following question remains still unanswered:

Do different classifiers identify the same defect-prone entities or different entities with the same defect-proneness?

In this paper we try to bridge this gap. We conjecture that the investigated classifiers are not equivalent even if they provide the same prediction accuracy. In other words, we believe that different classifiers could be able to identify different defect-prone entities. This calls for a combined approach that employs different (and complementary) classifiers aiming at improving the prediction accuracy of stand-alone approaches.

To verify our conjecture we conducted an empirical study aiming at statistically analyzing the equivalence of different classifiers. The study is based on Principal Component Analysis (PCA) and overlap metrics. With such analyses we are able to identify classifiers that complement each others, *i.e.,* they identify different sets of defect-prone entities. The experimentation was conducted on 10 open-source software systems from the Promise repository. In the context of our study we investigated six different machine learning techniques used for defect prediction in many previous works [5], [6], [7], [8], [9], [2]. These techniques belong to four different categories: regression functions, neural networks, decision trees, and rules models. All these approaches were experimented in the context of cross-project defect prediction, that represents one of the main challenges in the defect prediction field [10].

The achieved results confirmed our conjecture. *The investigated classifiers are not equivalent despite the similar overall prediction accuracy. This means that they are able to identify different sets of defect-prone entities*. This is confirmed by our statistical analysis but especially by the superior performances of a novel combined approach, coined as CODEP (**CO**mbined **DE**fect **P**redictor), that uses machine learning techniques to combine different and complementary classifiers.

In summary, the contributions of this paper are:

Classifiers are highly complementary

Different classifiers capture different sets of buggy components!

2014

The impact of Classifiers

2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models

Baljinder Ghotra, Shane McIntosh, Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University, Canada
{ghotra, mcintosh, ahmed}@cs.queensu.ca

***Abstract*—Defect prediction models help software quality assurance teams to effectively allocate their limited resources to the most defect-prone software modules. A variety of classification techniques have been used to build defect prediction models ranging from simple (e.g., logistic regression) to advanced techniques (e.g., Multivariate Adaptive Regression Splines (MARS)). Surprisingly, recent research on the NASA dataset suggests that the performance of a defect prediction model is not significantly impacted by the classification technique that is used to train it. However, the dataset that is used in the prior study is both: (a) noisy, i.e., contains erroneous entries and (b) biased, i.e., only contains software developed in one setting. Hence, we set out to replicate this prior study in two experimental settings. First, we apply the replicated procedure to the same (known-to-be noisy) NASA dataset, where we derive similar results to the prior study, i.e., the impact that classification techniques have appear to be minimal. Next, we apply the replicated procedure to two new datasets: (a) the cleaned version of the NASA dataset and (b) the PROMISE dataset, which contains open source software developed in a variety of settings (e.g., Apache, GNU). The results in these new datasets show a clear, statistically distinct separation of groups of techniques, i.e., the choice of classification technique has an impact on the performance of defect prediction models. Indeed, contrary to earlier research, our results suggest that some classification techniques tend to produce defect prediction models that outperform others.**

I. Introduction

A disproportionate amount of the cost of developing software is spent on maintenance [15]. Fixing defects is a central software maintenance activity. However, before defects can be fixed, they must be detected. Software Quality Assurance (SQA) teams are dedicated to this task of defect detection.

Defect prediction models can be used to assist SQA teams with defect detection. Broadly speaking, defect prediction models are trained using software metrics (e.g., size and complexity metrics) to predict whether software modules will be defective or not in the future. Knowing which software modules are likely to be defect-prone before a system has been deployed allows practitioners to more efficiently and effectively allocate SQA effort.

To perform defect prediction studies [21, 58], researchers have explored the use of various classification techniques to train defect prediction models [12, 34]. For example, in early studies, researchers used simple techniques like logistic regression [2, 7, 47] and linear regression [24, 65] to train defect prediction models. In more recent work, researchers have used more advanced techniques like Multivariate Adaptive Regression Splines (MARS) [5, 27], Personalized Change Classification (PCC) [27], and Logistic Model Trees (LMT) [51, 53]. Ensemble methods that combine different machine learning techniques have also been explored [14, 61]. Moreover, researchers have started to develop context-sensitive techniques that are aware of the peculiarities of software defects [5, 35].

Despite recent advances in machine learning, studies suggest that the classification technique used to train defect prediction models has little impact on its performance [34, 38, 56]. For example, Lessmann *et al.* [34] conducted a study comparing the performance of 22 different classification techniques on the NASA corpus. Their results show that the performance of 17 of the 22 classification techniques are statistically indistinguishable from each other.

However, the NASA corpus that was used in the prior work is noisy and biased. Indeed, Shepperd *et al.* [57] found that the original NASA corpus contains several erroneous and implausible entries. This noise in the studied corpus may have led prior studies to draw incorrect conclusions. Furthermore, the studied corpus only contains proprietary software developed within one organization. Software developed in a different setting (e.g., open source) may lead to different conclusions.

Therefore, we set out to revisit the findings of Lessmann *et al.* [34] both in the original, known-to-be noisy setting (Section IV), and in two additional settings (Section V). We find that we can indeed replicate the initial results of Lessmann *et al.* [34] when the procedure is reapplied to the known-to-be noisy NASA corpus. On the other hand, the results of our experiment in two additional settings seem to contradict Lessmann *et al.*'s early results. We structure our extended replication by addressing the following two research questions:

RQ1 Do the defect prediction models of different classification techniques still perform similarly when trained using the cleaned NASA corpus?
No, we find a clear separation of classification techniques when models are trained using the cleaned NASA corpus [57]. The Scott-Knott statistical test [26] groups the classification techniques into four statistically distinct ranks. Our results show that models trained using LMT and statistical techniques like Simple Logistic tend to outperform models trained using clustering, rule-based, Support Vector Machines (SVM), nearest neighbour, and neural network techniques.

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.91

The choice of machine learners strongly impact the performance!

The performances can increase o decrease up to 30% depending on the types of classification applied.

2015

How to make a choice?

Ensemble Techniques

Classification accuracy

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \qquad [0,1]$$

Type A

Type B

$$\text{Accuracy} = \frac{24}{25} = 0.96\ (96\%)$$

Confusion matrix

	Predicted A	Predicted B
Actual A	**TN=126**	**FP=13**
Actual B	**FN=24**	**TP=60**

Accuracy = 0.83

Confusion matrix: Precision

Precision

	Predicted A	Predicted B
Actual A	**TN=126**	**FP=13**
Actual B	**FN=24**	**TP=60**

Accuracy = 0.83

$$\text{Precision} = \frac{\mathbf{TP}}{\mathbf{TP+FP}} = \frac{\mathbf{T(B)}}{\mathbf{T(B)+F(B)}} = \frac{\mathbf{60}}{\mathbf{60+13}} = 0.82$$

Confusion matrix: Precision

For a spam filter we want to minimize FP or maximize Precision

Precision

	Predicted OK	Predicted Spam
Actual OK	**TN=126**	**FP=13**
Actual Spam	**FN=24**	**TP=60**

Accuracy = 0.83

$$\text{Precision} = \frac{\mathbf{TP}}{\mathbf{TP+FP}} = \frac{\mathbf{T(spam)}}{\mathbf{T(s)+F(not)}} = \frac{\mathbf{60}}{\mathbf{60+13}} = 0.82$$

Confusion matrix: Recall

	Predicted A	Predicted B
Actual A	**TN=126**	**FP=13**
Actual B	**FN=24**	**TP=60**

Recall

Accuracy = 0.83

$$\text{Recall} = \frac{\mathbf{TP}}{\mathbf{TP+FN}} = \frac{\mathbf{T(B)}}{\mathbf{T(B)+F(A)}} = \frac{\mathbf{60}}{\mathbf{60+24}} = 0.71$$

Confusion matrix: Recall

For a cancer detector we want to minimize FN or maximize Recall

	Predicted OK	Predicted cancer
Actual OK	**TN=126**	**FP=13**
Actual cancer	**FN=24**	**TP=60**

Recall

Accuracy = 0.83

$$\text{Recall} = \frac{\mathbf{TP}}{\mathbf{TP+FN}} = \frac{\mathbf{T(cancer)}}{\mathbf{T(c)+F(not)}} = \frac{\mathbf{60}}{\mathbf{60+24}} = 0.71$$

cancer

Confusion matrix: F-Measure

Precision

	Predicted A	Predicted B
Actual A	**TN=126**	**FP=13**
Actual B	**FN=24**	**TP=60**

Recall

Accuracy = 0.83

Precision = 0.82

Recall = 0.71

$$\text{F-Measure} = \mathbf{2} * \frac{\textbf{Precision} * \textbf{Recall}}{\textbf{Precision} + \textbf{Recall}} = \frac{\mathbf{2*0.82*0.71}}{\mathbf{0.82+0.71}} = 0.86$$

Matthews Correlation Coefficient

	Predicted A	Predicted B
Actual A	**TN=126**	**FP=13**
Actual B	**FN=24**	**TP=60**

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \qquad [-1,1]$$

Matthews Correlation Coefficient

	Predicted A	Predicted B
Actual A	**TN=0**	**FP=5**
Actual B	**FN=0**	**TP=95**

$$\text{F-Measure} = \frac{\mathbf{2 * TP}}{\mathbf{2 * TP + FP + FN}} = \frac{\mathbf{2 * 95}}{\mathbf{2 * 95 + 5}} = 0.97$$

$$\text{MCC} = \frac{\mathbf{TP * TN - FP * FN}}{\sqrt{\mathbf{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}} = \text{undef.}$$

Matthews Correlation Coefficient

	Predicted A	Predicted B
Actual A	**TN=1**	**FP=4**
Actual B	**FN=5**	**TP=90**

$$\text{F-Measure} = \frac{\mathbf{2 * TP}}{\mathbf{2 * TP + FP + FN}} = \frac{\mathbf{2 * 90}}{\mathbf{2 * 90 + 4 + 5}} = 0.95$$

$$\text{MCC} = \frac{\mathbf{TP * TN - FP * FN}}{\sqrt{\mathbf{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}} = 0.135$$

Matthews Correlation Coefficient

	Predicted A	Predicted B
Actual A	**TN=90**	**FP=5**
Actual B	**FN=4**	**TP=1**

$$\text{F-Measure} = \frac{\mathbf{2 * TP}}{\mathbf{2 * TP + FP + FN}} = \frac{\mathbf{2 * 1}}{\mathbf{2 * 1 + 5 + 4}} = 0.18$$

$$\text{MCC} = \frac{\mathbf{TP * TN - FP * FN}}{\sqrt{\mathbf{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}} = 0.135$$

Hands-on: mining software repositories

Luca Pascarella and Fabio Palomba

PyDriller: A Python framework

PyDriller: Python Framework for Mining Software Repositories

Davide Spadini
Delft University of Technology
Software Improvement Group
Delft, The Netherlands
d.spadini@sig.eu

Maurício Aniche
Delft University of Technology
Delft, The Netherlands
m.f.aniche@tudelft.nl

Alberto Bacchelli
University of Zurich
Zurich, Switzerland
bacchelli@ifi.uzh.ch

ABSTRACT

Software repositories contain historical and valuable information about the overall development of software systems. Mining software repositories (MSR) is nowadays considered one of the most interesting growing fields within software engineering. MSR focuses on extracting and analyzing data available in software repositories to uncover interesting, useful, and actionable information about the system. Even though MSR plays an important role in software engineering research, few tools have been created and made public to support developers in extracting information from Git repository. In this paper, we present PyDriller, a Python Framework that eases the process of mining Git. We compare our tool against the state-of-the-art Python Framework GitPython, demonstrating that PyDriller can achieve the same results with, on average, 50% less LOC and significantly lower complexity.

URL: https://github.com/ishepard/pydriller,
Materials: https://doi.org/10.5281/zenodo.1327363,
Pre-print: https://doi.org/10.5281/zenodo.1327411

CCS CONCEPTS

• **Software and its engineering**;

KEYWORDS

Mining Software Repositories, GitPython, Git, Python

ACM Reference Format:
Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python Framework for Mining Software Repositories. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3236024.3264598

1 INTRODUCTION

Mining software repository (MSR) techniques allow researchers to analyze the information generated throughout the software development process, such as source code, version control systems metadata, and issue reports [5, 18, 22]. With such analysis, researches can empirically investigate, understand, and uncover useful and actionable insights for software engineering, such as understanding the impact of code smells [13–15], exploring how developers are doing code reviews [2, 4, 10, 21] and which testing practices they follow [20], predicting classes that are more prone to change/defects [3, 6, 16, 17], and identifying the core developers of a software team to transfer knowledge [12].

Among the different sources of information researchers can use, version control systems, such as Git, are among the most used ones. Indeed, version control systems provide researchers with precise information about the source code, its evolution, the developers of the software, and the commit messages (which explain the reasons for changing).

Nevertheless, extracting information from Git repositories is not trivial. Indeed, many frameworks can be used to interact with Git (depending on the preferred programming language), such as GitPython [1] for Python, or JGit for Java [8]. However, these tools are often difficult to use. One of the main reasons for such difficulty is that they encapsulate all the features from Git, hence, developers are forced to write long and complex implementations to extract even simple data from a Git repository.

In this paper, we present PyDriller, a Python framework that helps developers to mine software repositories. PyDriller provides developers with simple APIs to extract information from a Git repository, such as commits, developers, modifications, diffs, and source code. Moreover, as PyDriller is a framework, developers can further manipulate the extracted data and quickly export the results to their preferred formats (*e.g.*, CSV files and databases).

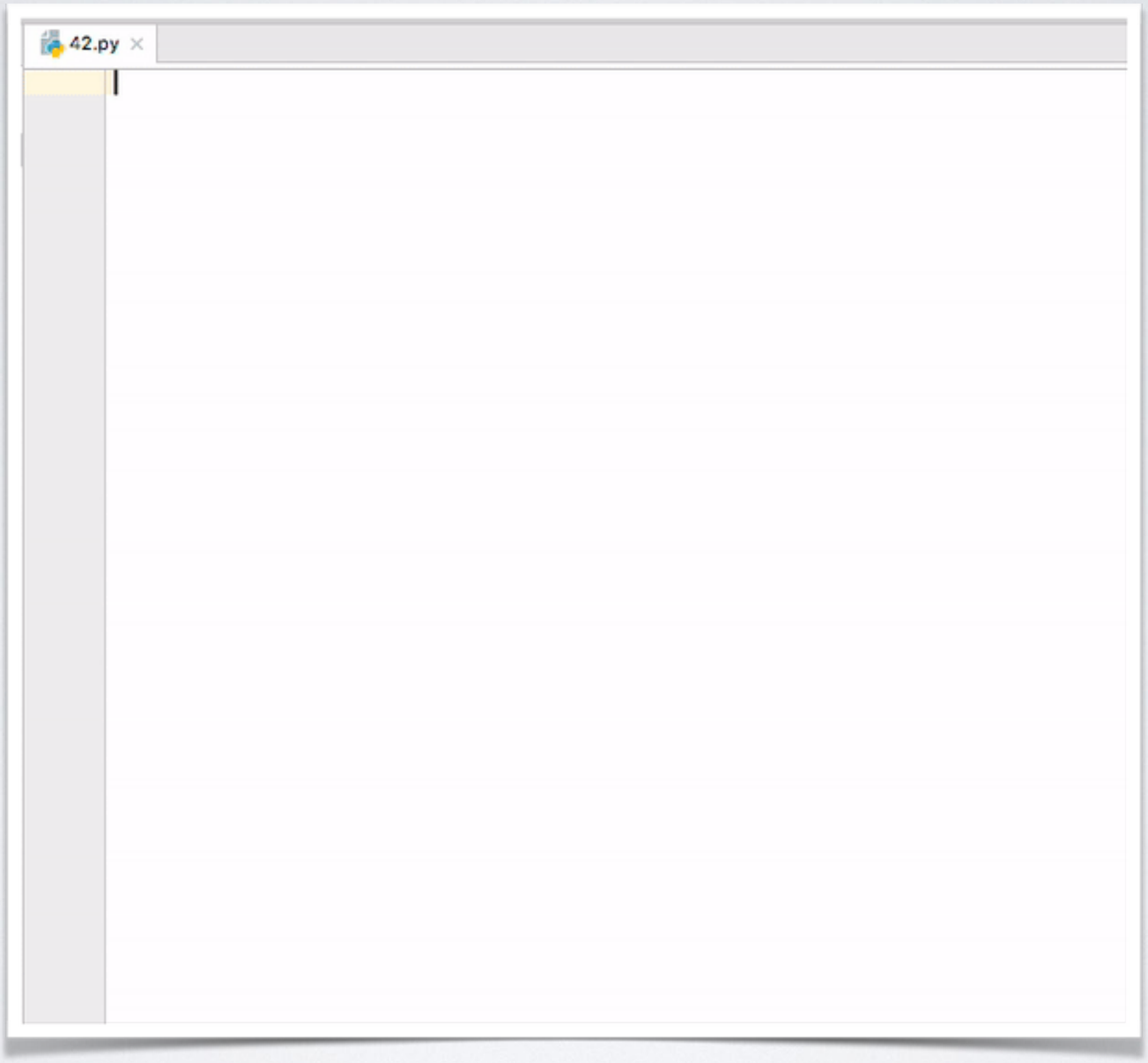
To evaluate the usefulness of our tool, we compare it with the state-of-the-art Python framework GitPython, in terms of implementation complexity, performance, and memory consumption. Our results show that PyDriller requires significantly fewer lines of code to perform the same task when compared to GitPython, with only a small drop in performance. Also, we asked six developers to perform tasks with both tools and found that all developers spend less time in learning and implementing tasks in PyDriller.

PyDriller is a wrapper of GitPython that eases MSR processes

- Traverse commits
- Product metrics
- Bug inducing commits

2018

PyDriller: A Python framework



https://github.com/ishepard/pydriller

Print commit HASHs

```
pip install pydriller
```

```
from pydriller import RepositoryMining

repo = 'https://github.com/SERG-Delft/jpacman-framework.git'

for commit in RepositoryMining(repo).traverse_commits():
    print(commit.hash)
```

https://github.com/ishepard/pydriller

Unique contributors

```
> mkdir Salerno
> cd Salerno
> git clone https://github.com/SERG-Delft/jpacman-framework.git
```

```
from pydriller import RepositoryMining

repo = '../Salerno/jpacman-framework'

authors = []
for commit in RepositoryMining(repo).traverse_commits():
    authors.append(commit.author.email)

unique_authors = set(authors)
print('Number of unique authors: {}'.format(len(unique_authors)))
print(unique_authors)
```

https://github.com/ishepard/pydriller

Product metrics

```
from pydriller import GitRepository

repo = '/Users/luca/TUProjects/Salerno/jpacman-framework'

gr = GitRepository(repo)

last_commit = gr.get_head()
print('Commit: {} changed {} files.'
    .format(last_commit.hash, len(last_commit.modifications)))
for mod in last_commit.modifications:
    print('File: {} has {} additions and {} deletions'
     .format(mod.filename, mod.added, mod.removed))
    print('Complexity: {}. LOC {}'
     .format(mod.complexity, mod.nloc))
```

Process metrics

```
from pydriller import RepositoryMining
from pydriller.domain.commit import ModificationType

repo = '/Users/luca/TUProjects/Salerno/jpacman-framework'
start = 'f3178b8'
stop = '51f041d'

files = {}
for commit in RepositoryMining(repo, from_commit=start,
to_commit=stop).traverse_commits():
    for mod in commit.modifications:
        if mod.filename.endswith('.java') and mod.change_type is
not ModificationType.DELETE:
            path = mod.new_path
            if path not in files:
                files[path] = []
            files.get(path, []).append(mod.change_type)

for file in files:
    print('Name: {}. Changes: {}'.format(file, len(files[file])))
```

Product metrics: Improved

```
from pydriller import RepositoryMining
from pydriller.domain.commit import ModificationType

repo = '/Users/luca/TUProjects/Salerno/jpacman-framework'
start = 'f3178b8'
stop = '51f041d'

files = {}
for commit in RepositoryMining(repo, from_commit=start, to_commit=stop).traverse_commits():
    for mod in commit.modifications:
        if mod.filename.endswith('.java') and mod.change_type is not ModificationType.DELETE:
            process_metrics = {'change': mod.change_type, 'added': mod.added, 'removed': mod.removed, 'loc':
mod.nloc, 'comp': mod.complexity}

            path = mod.new_path
            if path not in files:
                files[path] = []
            files.get(path, []).append(process_metrics)

output = open('output.csv', 'w')
output.write('file,n-changes,added,removed,loc,complexity\n'.format())

for key, value in files.items():
    n_changes = len(value)
    last_added = value[n_changes - 1]['added']
    last_removed = value[n_changes - 1]['removed']
    last_loc = value[n_changes - 1]['loc']
    last_comp = value[n_changes - 1]['comp']
    print('Changes: {}. Added: {}. Removed: {}. LOC: {}. Comp.: {}. File: {}'.format(n_changes, last_added,
last_removed, last_loc, last_comp, key))
    # Append process metrics to CSV file
    output.write('{},{},{},{},{},{}\n'.format(key, n_changes, last_added, last_removed, last_loc,
last_comp))
```

Bug inducing commits

```
from pydriller import RepositoryMining, GitRepository

repo = '/Users/luca/TUProjects/Salerno/jpacman-framework'

fix_commits = []
for commit in RepositoryMining(repo,
only_modifications_with_file_types=['.java']).traverse_commits():
    if 'fix' in commit.msg:
        fix_commits.append(commit)

gr = GitRepository(repo)

buggy_commit_hashs = set()
for fix_commit in fix_commits:
    bug_commits = gr.get_commits_last_modified_lines(fix_commit)
    buggy_commit_hashs.update(bug_commits)  # Add a set to a set

print('Number of commits with fixes: {}'.format(len(fix_commits)))
print('Number of commits with bugs: {}'.format(len(buggy_commit_hashs)))
```

Simple defect predictor

```
repo = '/Users/luca/TUProjects/Salerno/jpacman-framework'

def get_buggy_commits():
    fix_commits = []
    for commit in RepositoryMining(repo).traverse_commits():
        if 'fix' in commit.msg:
            fix_commits.append(commit)
    gr = GitRepository(repo)
    buggy_commit_hashs = set()
    for fix_commit in fix_commits:
        bug_commits = gr.get_commits_last_modified_lines(fix_commit)
        buggy_commit_hashs.update(bug_commits)  # Add a set to a set
    return buggy_commit_hashs

def calculate_metrics(bugs):
    files = {}
    for commit in RepositoryMining(repo).traverse_commits():
        for mod in commit.modifications:
            if mod.filename.endswith('.java') and mod.change_type is not ModificationType.DELETE:
                buggy = True if commit.hash in bugs else False
                process_metrics = {'change': mod.change_type, 'added': mod.added, 'removed': mod.removed,
'loc': mod.nloc, 'comp': mod.complexity, 'buggy': buggy}
                path = mod.new_path
                if path not in files:
                    files[path] = []
                files.get(path, []).append(process_metrics)
    return files

# Main
buggy_commit = get_buggy_commits()
files = calculate_metrics(buggy_commit)
```

Machine Learning Software in Java

Weka: Practical Machine Learning Tools and Techniques with Java Implementations

Ian H. Witten, Eibe Frank, Len Trigg, Mark Hall, Geoffrey Holmes, and Sally Jo Cunningham,
Department of Computer Science, University of Waikato, New Zealand.

Introduction

The Waikato Environment for Knowledge Analysis (Weka) is a comprehensive suite of Java class libraries that implement many state-of-the-art machine learning and data mining algorithms. Weka is freely available on the World-Wide Web and accompanies a new text on data mining [1] which documents and fully explains all the algorithms it contains. Applications written using the Weka class libraries can be run on any computer with a Web browsing capability; this allows users to apply machine learning techniques to their own data regardless of computer platform.

Tools are provided for pre-processing data, feeding it into a variety of learning schemes, and analyzing the resulting classifiers and their performance. An important resource for navigating through Weka is its on-line documentation, which is automatically generated from the source.

The primary learning methods in Weka are "classifiers", and they induce a rule set or decision tree that models the data. Weka also includes algorithms for learning association rules and clustering data. All implementations have a uniform command-line interface. A common evaluation module measures the relative performance of several learning algorithms over a given data set.

Tools for pre-processing the data, or "filters," are

Figure 1. They provide interfaces to pre-processing routines including feature selection, classifiers for both categorical and numeric learning tasks, meta-classifiers for enhancing the performance of classifiers (for example, boosting and bagging), evaluation according to different criteria (for example, accuracy, entropy, root-squared mean error, cost-sensitive classification, etc.) and experimental support for verifying the robustness of models (cross-validation, bias-variance decomposition, and calculation of the margin).

Weka's core

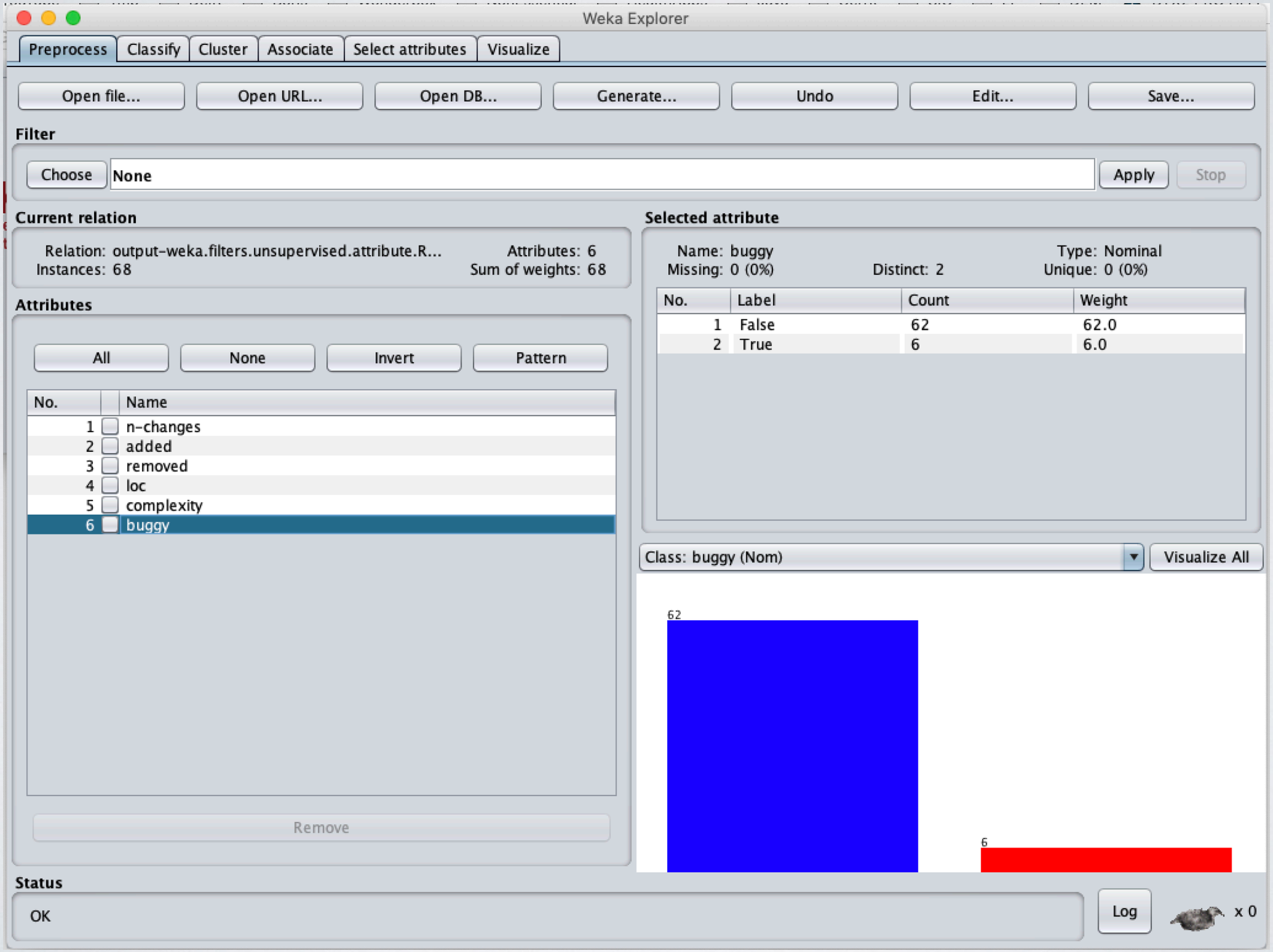
The *core* package contains classes that are accessed from almost every other class in Weka. The most important classes in it are *Attribute*, *Instance*, and *Instances*. An object of class Attribute represents an attribute—it contains the attribute's name, its type, and, in case of a nominal attribute, its possible values. An object of class Instance contains the attribute values of a particular instance; and an object of class Instances contains an ordered set of instances—in other words, a dataset.

Data Pre-Processing

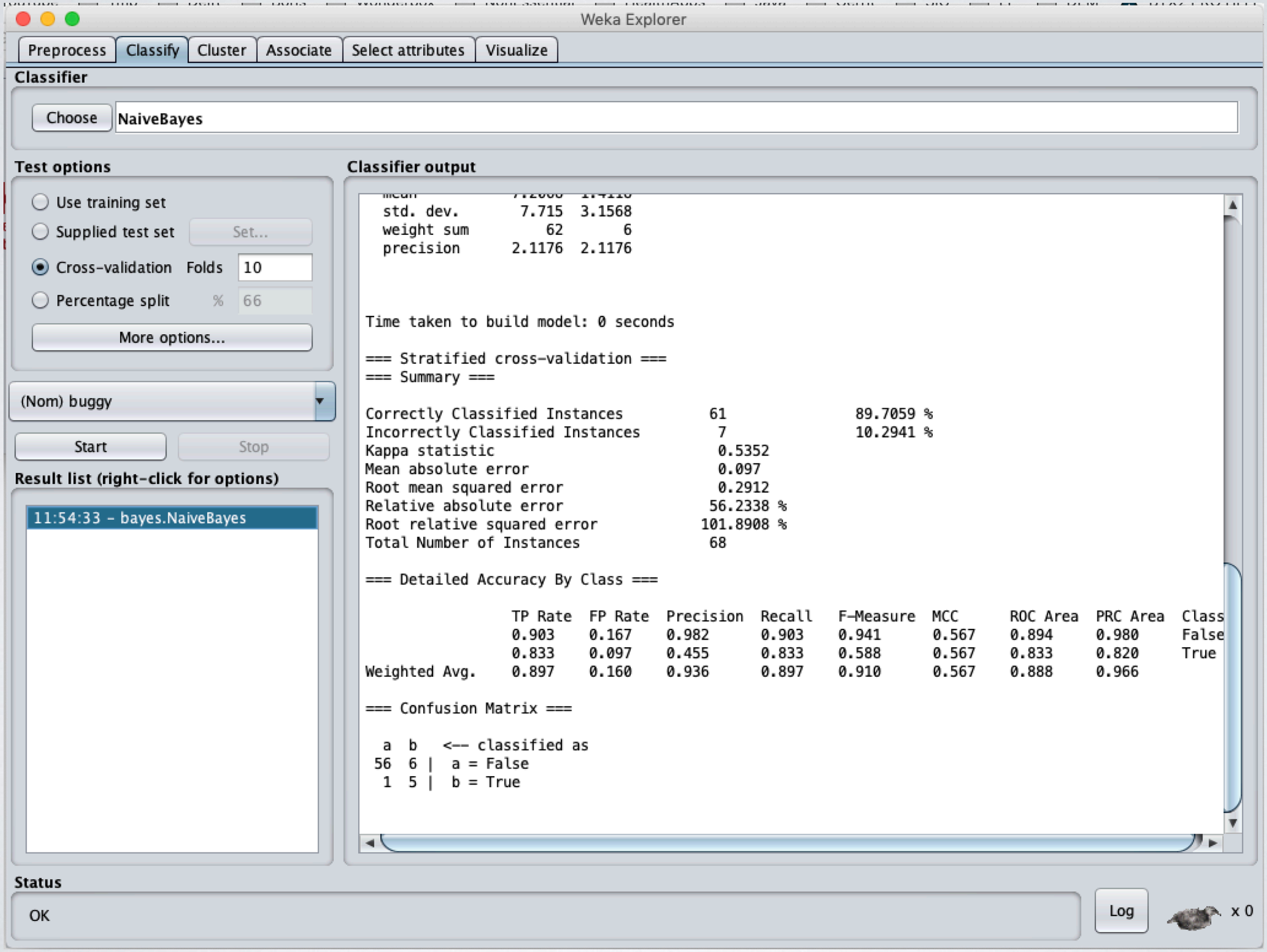
Weka's pre-processing capability is encapsulated in an extensive set of routines, called *filters*, that enable data to be processed at the instance and attribute value levels. Table 1 lists the most important filter algorithms that are included.

https://www.cs.waikato.ac.nz/ml/weka/

Weka


Weka Explorer
Preprocess
Classify
Cluster
Associate
Select attributes
Visualize
Open file...
Open URL...
Open DB...
Generate...
Undo
Edit...
Save...
Filter
Choose
None
Apply
Stop
Current relation
Relation: output-weka.filters.unsupervised.attribute.R...
Instances: 68
Attributes: 6
Sum of weights: 68
Attributes
All
None
Invert
Pattern
No.
Name
1 n-changes
2 added
3 removed
4 loc
5 complexity
6 buggy
Remove
Selected attribute
Name: buggy
Missing: 0 (0%)
Distinct: 2
Type: Nominal
Unique: 0 (0%)
No.
Label
Count
Weight
1 False 62 62.0
2 True 6 6.0
Class: buggy (Nom)
Visualize All
62
6
Status
OK
Log
x 0

Weka


Weka Explorer
Preprocess
Classify
Cluster
Associate
Select attributes
Visualize
Classifier
Choose
NaiveBayes
Test options
Use training set
Supplied test set
Set...
Cross-validation
Folds
10
Percentage split
%
66
More options...
Classifier output
std. dev. 7.715 3.1568
weight sum 62 6
precision 2.1176 2.1176
Time taken to build model: 0 seconds
=== Stratified cross-validation ===
=== Summary ===
Correctly Classified Instances 61 89.7059 %
Incorrectly Classified Instances 7 10.2941 %
Kappa statistic 0.5352
Mean absolute error 0.097
Root mean squared error 0.2912
Relative absolute error 56.2338 %
Root relative squared error 101.8908 %
Total Number of Instances 68
=== Detailed Accuracy By Class ===
TP Rate FP Rate Precision Recall F-Measure MCC ROC Area PRC Area Class
0.903 0.167 0.982 0.903 0.941 0.567 0.894 0.980 False
0.833 0.097 0.455 0.833 0.588 0.567 0.833 0.820 True
Weighted Avg. 0.897 0.160 0.936 0.897 0.910 0.567 0.888 0.966
=== Confusion Matrix ===
a b <-- classified as
56 6 | a = False
1 5 | b = True
(Nom) buggy
Start
Stop
Result list (right-click for options)
11:54:33 - bayes.NaiveBayes
Status
OK
Log
x 0