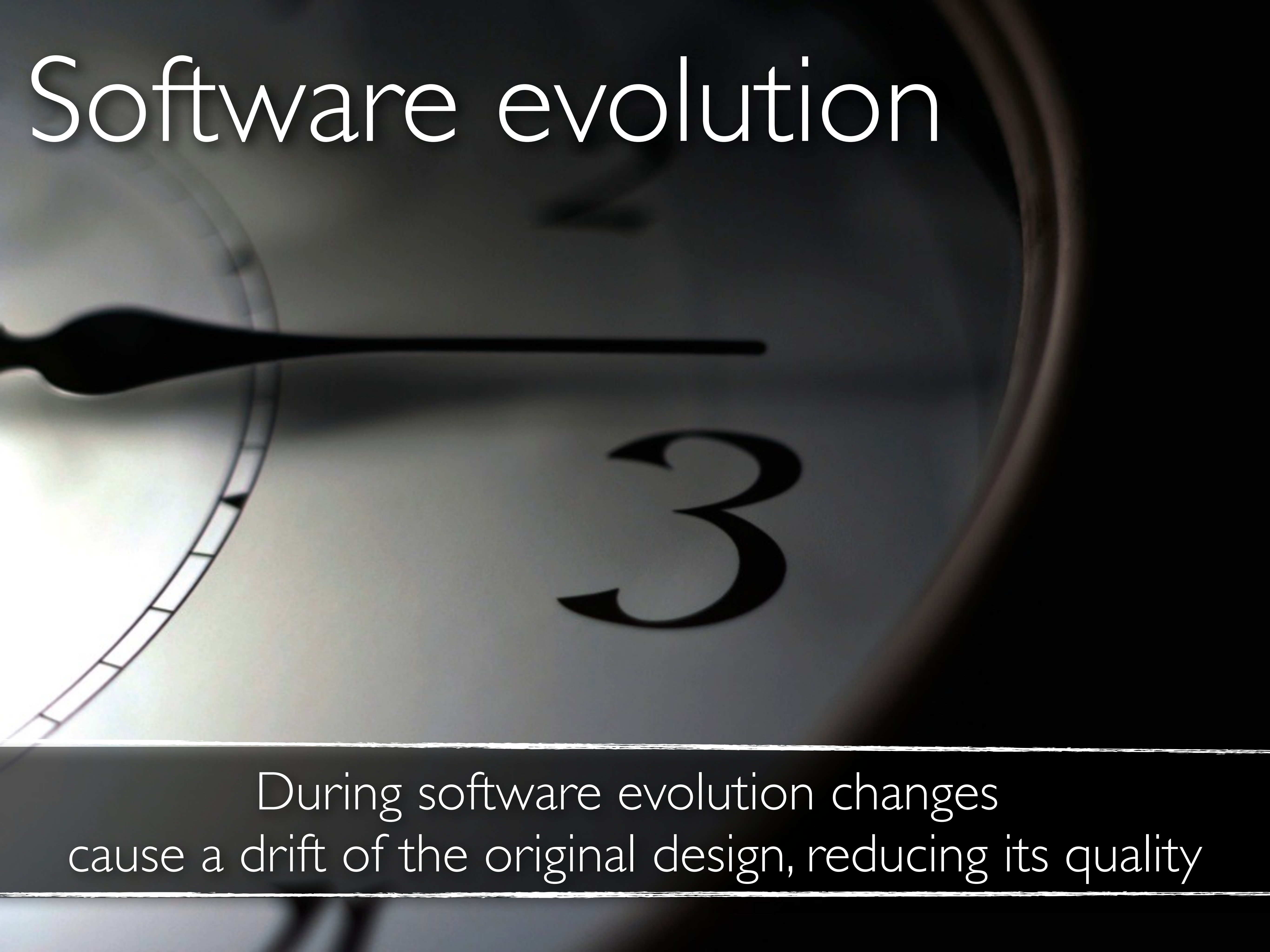
Code and Test Smells
Understanding and Detecting Them

Fabio Palomba
Assistant Professor
University of Salerno (Italy)
https://fpalomba.github.io

Code and Test **Smells**

Understanding and Detecting Them

Software evolution



During software evolution changes
cause a drift of the original design, reducing its quality

Low design quality …

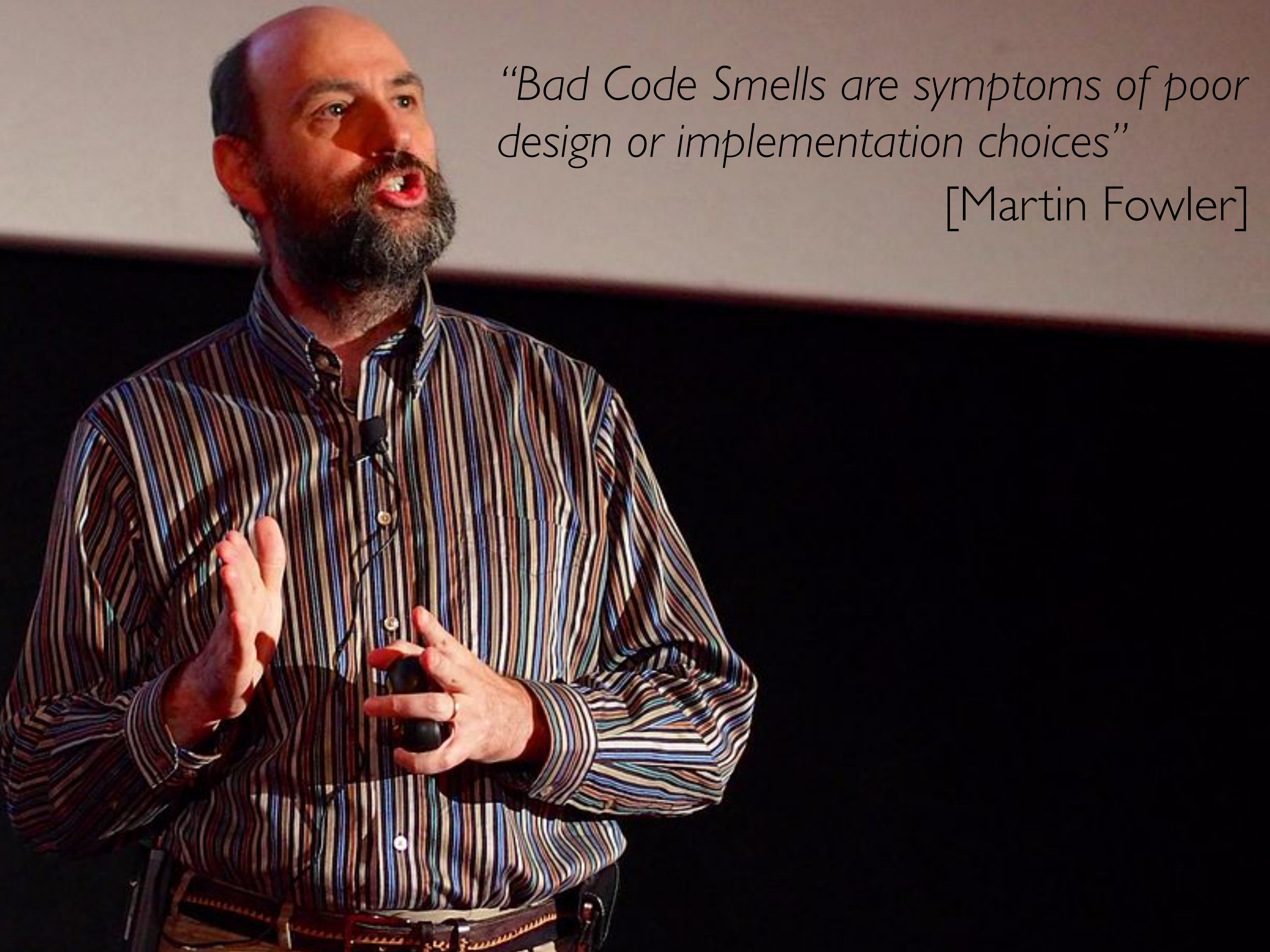
… has been associated with lower productivity, greater rework, and more significant efforts for developers

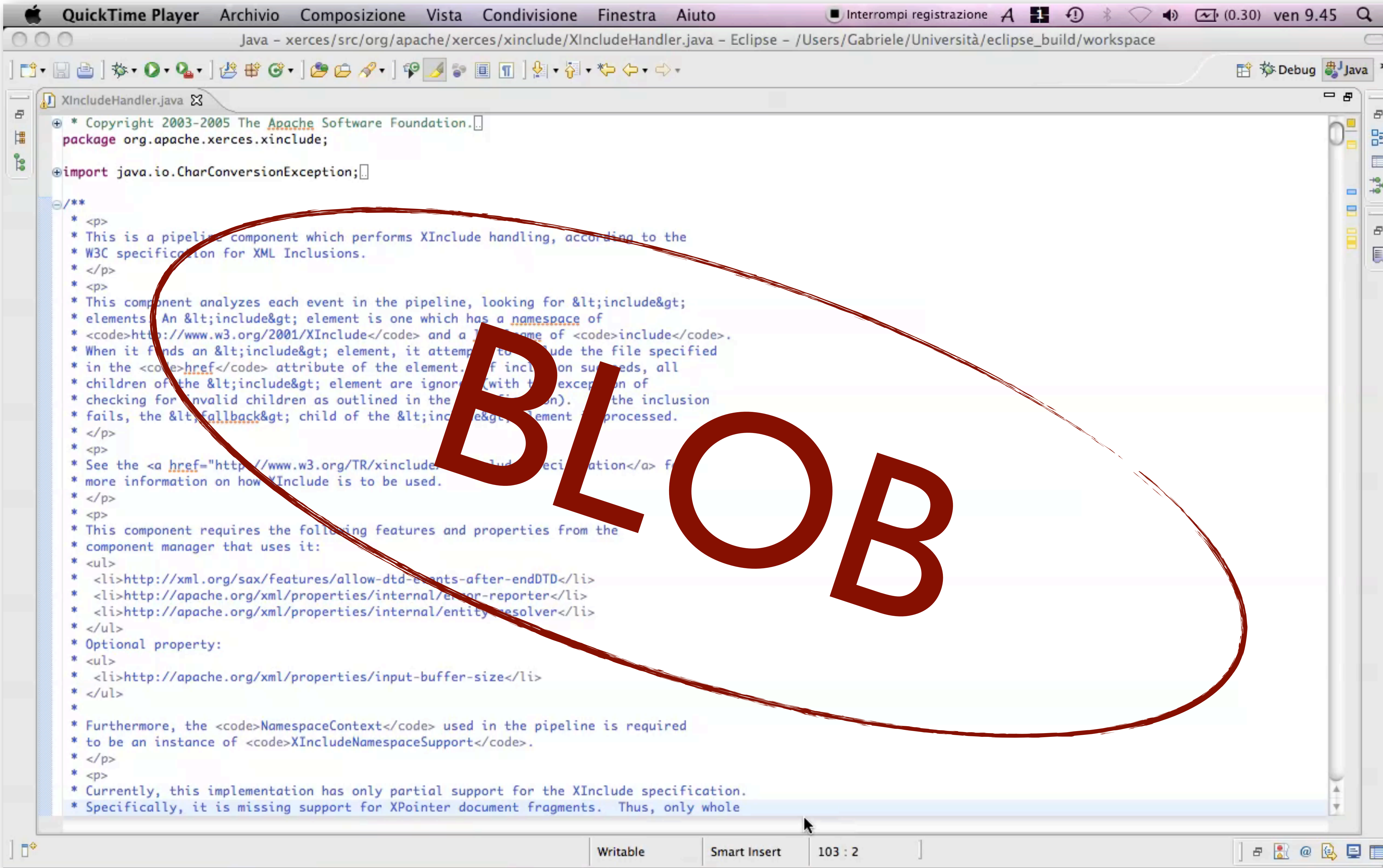
Victor R. Basili, Lionel Briand, and Walcelio L. Melo. A Validation Of Object-Oriented Design Metrics As Quality Indicators. IEEE Transactions on Software Engineering (TSE), 22(10):751–761, 1995.

Aaron B. Binkley and Stephen R. Schach. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. 20th International Conference on Software Engineering (ICSE 1998), pages 452–455.

Lionel C. Briand, Juergen Wuest, and Hakim Lounis. Using Coupling Measurement for Impact Analysis in Object-Oriented Systems. 15th IEEE International Conference on Software Maintenance (ICSM 1999), pages 475–482.

Lionel C. Briand, Jurgen Wust, Stefan V. Ikonomovski, and Hakim Lounis. Investigating quality factors in object-oriented designs: an industrial case study. 21st International Conference on Software Engineering (ICSE 1999), pages 345–354.


"Bad Code Smells are symptoms of poor design or implementation choices"
[Martin Fowler]


QuickTime Player Archivio Composizione Vista Condivisione Finestra Aiuto
Interrompi registrazione
(0.30) ven 9.45
Java – xerces/src/org/apache/xerces/xinclude/XIncludeHandler.java – Eclipse – /Users/Gabriele/Università/eclipse_build/workspace
Debug
Java
XIncludeHandler.java
* Copyright 2003-2005 The Apache Software Foundation.
package org.apache.xerces.xinclude;
import java.io.CharConversionException;
/**
* <p>
* This is a pipeli component which performs XInclude handling, acc ing to the
* W3C specifica on for XML Inclusions.
* </p>
* <p>
* This comp nent analyzes each event in the pipeline, looking for <include>
* elements. An <include> element is one which has a namespace of
* <code>htt ://www.w3.org/2001/XInclude</code> and a me of <code>include</code>.
* When it f nds an <include> element, it attem ude the file specified
* in the <co e>href</code> attribute of the element. f inc on su eds, all
* children of the <include> element are ignor (with exce n of
* checking for nvalid children as outlined in the n). the inclusion
* fails, the < fallback> child of the <inc ement processed.
* </p>
* <p>
* See the <a href="http //www.w3.org/TR/xinclude eci ation f
* more information on how XInclude is to be used.
* </p>
* <p>
* This component requires the foll ing features and properties from the
* component manager that uses it:
*
* http://xml.org/sax/features/allow-dtd- nts-after-endDTD
* http://apache.org/xml/properties/internal/ or-reporter
* http://apache.org/xml/properties/internal/entit esolver
*
* Optional property:
*
* http://apache.org/xml/properties/input-buffer-size
*
*
* Furthermore, the <code>NamespaceContext</code> used in the pipeline is required
* to be an instance of <code>XIncludeNamespaceSupport</code>.
* </p>
* <p>
* Currently, this implementation has only partial support for the XInclude specification.
* Specifically, it is missing support for XPointer document fragments. Thus, only whole
Writable
Smart Insert
103 : 2
BLOB

Blob (or God Class)

A Blob (also named God Class) is a "class implementing several responsibilities, having a large number of attributes, operations and dependencies with data classes".

[Martin Fowler]

Blob (or God Class)

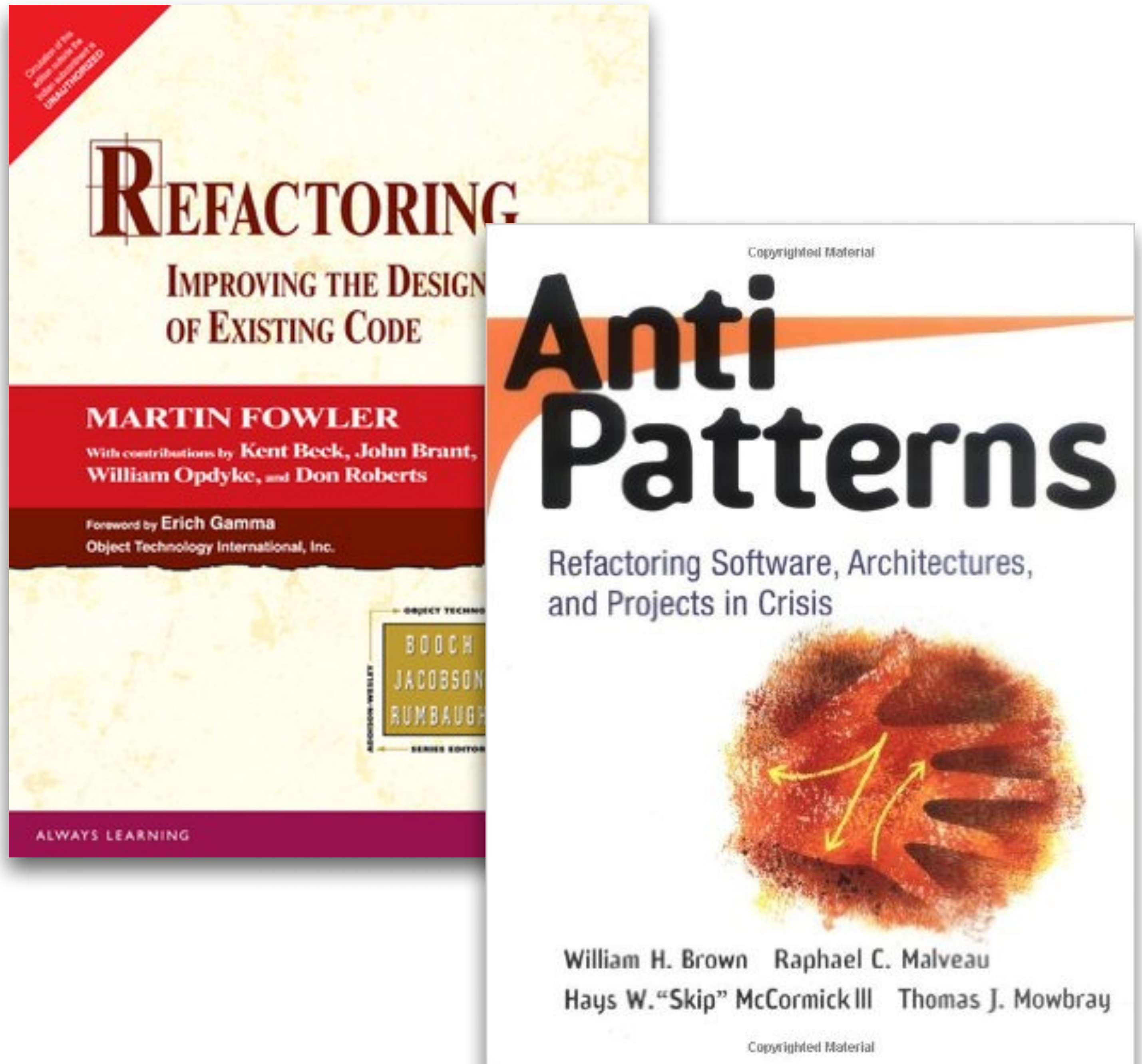
A Blob (also named God Class) is a "class implementing several responsibilities, having a large number of attributes, operations and dependencies with data classes".

[Martin Fowler]

Consequences

Increasing maintenance costs due to the difficulty of comprehending and maintaining the class.

40+ different smells



40+ different smells
… and even more

Energy-related code smells

Security-related code smells

. . .

Performance-related code smells

40+ different smells
… and even more

Quality-related code smells

Negative Impact of Bad Smells

2011 15th European Conference on Software Maintenance and Reengineering

An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, On Program Comprehension

Marwen Abbes[1,3], Foutse Khomh[2], Yann-Gaël Guéhéneuc[3], Giuliano Antoniol[3]
[1] Dépt. d'Informatique et de Recherche Opérationnelle, Université de Montréal, Montréal, Canada
[2] Dept. of Elec. and Comp. Engineering, Queen's University, Kingston, Ontario, Canada
[3] Ptidej Team, SOCCER Lab, DGIGL, École Polytechnique de Montréal, Canada
E-mails: marwen.abbes@umontreal.ca, foutse.khomh@queensu.ca
yann-gael.gueheneuc@polymtl.ca, antoniol@ieee.org

***Abstract*—Antipatterns are "poor" solutions to recurring design problems which are conjectured in the literature to make object-oriented systems harder to maintain. However, little quantitative evidence exists to support this conjecture. We performed an empirical study to investigate whether the occurrence of antipatterns does indeed affect the understandability of systems by developers during comprehension and maintenance tasks. We designed and conducted three experiments, with 24 subjects each, to collect data on the performance of developers on basic tasks related to program comprehension and assessed the impact of two antipatterns and of their combinations: Blob and Spaghetti Code. We measured the developers' performance with: (1) the NASA task load index for their effort; (2) the time that they spent performing their tasks; and, (3) their percentages of correct answers. Collected data show that the occurrence of one antipattern does not significantly decrease developers' performance while the combination of two antipatterns impedes significantly developers. We conclude that developers can cope with one antipattern but that combinations of antipatterns should be avoided possibly through detection and refactorings.**

***Keywords*-Antipatterns, Blob, Spaghetti Code, Program Comprehension, Program Maintenance, Empirical Software Engineering.**

I. Introduction

Context: In theory, antipatterns are "poor" solutions to recurring design problems; they stem from experienced software developers' expertise and describe common pitfalls in object-oriented programming, *e.g.*, Brown's 40 antipatterns [1]. Antipatterns are generally introduced in systems by developers not having sufficient knowledge and–or experience in solving a particular problem or having misapplied some design patterns. Coplien [2] described an antipattern as "something that looks like a good idea, but which back-fires badly when applied". In practice, antipatterns relate to and manifest themselves as code smells in the source code, symptoms of implementation and–or design problems [3].

An example of antipattern is the Blob, also called God Class. The Blob is a large and complex class that centralises the behavior of a portion of a system and only uses other classes as data holders, *i.e.*, data classes. The main characteristic of a Blob class are: a large size, a low cohesion, some method names recalling procedural programming, and its association with data classes, which only provide fields and–or accessors to their fields. Another example of antipattern is the Spaghetti Code, which is characteristic of procedural thinking in object-oriented programming. Spaghetti Code classes have little structure, declare long methods with no parameters, and use global variables; their names and their methods names may suggest procedural programming. They do not exploit and may prevent the use of object-orientation mechanisms: polymorphism and inheritance.

Premise: Antipatterns are conjectured in the literature to decrease the quality of systems. Yet, despite the many studies on antipatterns summarised in Section II, few studies have empirically investigated the impact of antipatterns on program comprehension. Yet, program comprehension is central to an effective software maintenance and evolution [4]: a good understanding of the source code of a system is essential to allow its inspection, maintenance, reuse, and extension. Therefore, a better understanding of the factors affecting developers's comprehension of source code is an efficient and effective way to ease maintenance.

Goal: We want to gather quantitative evidence on the relations between antipatterns and program comprehension. In this paper, we focus on the system understandability, which is the degree to which the source code of a system can be easily understood by developers [5]. Gathering evidence on the relation between antipatterns and understandability is one more step [6] towards (dis)proving the conjecture in the literature about antipatterns and increasing our knowledge about the factors impacting program comprehension.

Study: We perform three experiments: we study whether systems with the antipattern Blob, first, and the Spaghetti Code, second, are more difficult to understand than systems without any antipattern. Third, we study whether systems with both Blob and Spaghetti Code are more difficult to understand than systems without any antipatterns. Each experiment is performed with 24 subjects and on three different systems developed in Java. The subjects are graduate students and professional developers with experience in software development and maintenance. We ask the subjects to perform three different program comprehension tasks covering three out of four categories

1534-5351/11 $26.00 © 2011 IEEE
DOI 10.1109/CSMR.2011.24

Bad Smells hinder code comprehensibility

[Abbes et al. CSMR 2011]

Negative Impact of Bad Smells

Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig

SOFTWARE COMPLEXITY AND MAINTENANCE COSTS

While the link between the difficulty in understanding computer software and the cost of maintaining it is appealing, prior empirical evidence linking software complexity to software maintenance costs is relatively weak [21]. Many of the attempts to link software complexity to maintainability are based on experiments involving small pieces of code, or are based on analysis of software written by students. Such evidence is valuable, but several researchers have noted that such results must be applied cautiously to the large-scale commercial application systems that account for most software maintenance expenditures [13, 17]. Furthermore, the limited large-scale research that has been undertaken has generated either conflicting results or none at all, as, for example, on the effects of software modularity and software structure [6, 12]. Additionally, none of the previous work develops estimates of the actual cost of complexity, estimates that could be used by software maintenance managers to make the best use of their resources. While research supporting the statistical significance of a factor is, of course, a necessary first step in this process, practitioners must also have an understanding of the practical magnitudes of the effects of complexity if they are to be able to make informed decisions.

This study analyzes the effects of software complexity on the costs of Cobol maintenance projects within a large commercial bank. It has been estimated that 60 percent of all business expenditures on computing are for maintenance of software written in Cobol [16]. Since over 50 billion lines of Cobol are estimated to exist worldwide, this also suggests that their maintenance represents an information systems (IS) activity of considerable economic importance. Using a previously developed economic model of software maintenance as a vehicle [2], this research estimates the impact of software complexity on the costs of software maintenance projects in a traditional IS environment. The model employs a multidimensional approach to measuring software complexity, and it controls for additional project factors under managerial control that are believed to affect maintenance project costs.

The analysis confirms that software maintenance costs are significantly affected by software complexity, measured in three dimensions: module size, procedure size, and branching complexity. The findings presented here also help to resolve the current debate over the functional form of the relationship between software complexity and the cost of software maintenance. The analysis further provides actual dollar estimates of the magnitude of this impact at a typical commercial site. The estimated costs are high enough to justify strong efforts on the part of software managers to monitor and control complexity. This analysis could also be used to assess the costs and benefits of a class of computer-aided software engineering (CASE) tools known as restructurers.

Previous Research and Conceptual Model

Software maintenance and complexity. This research adopts the ANSI/IEEE standard 729 definition of maintenance: modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment [28]. Research on the costs of software maintenance has much in common with research on the costs of new software development, since both involve the creation of working code through the efforts of human developers equipped with appropriate experience, tools, and techniques. Software maintenance, however, is fundamentally different from new systems development in that the soft-

Bad Smells increase maintenance costs

[Banker et al. Communications of the ACM]

Negative Impact of Bad Smells

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, Départment 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

Springer

Bad Smells increase change- and fault-proneness

[Khomh et al. EMSE 2012]

Evaluating the Lifespan of Code Smells using Software Repository Mining

Ralph Peters
Delft University of Technology
The Netherlands
Email: ralphpeters85@gmail.com

Andy Zaidman
Delft University of Technology
The Netherlands
Email: a.e.zaidman@tudelft.nl

***Abstract*—An anti-pattern is a commonly occurring solution to a recurring problem that will typically negatively impact code quality. Code smells are considered to be symptoms of anti-patterns and occur at source code level. The lifespan of code smells in a software system can be determined by mining the software repository on which the system is stored. This provides insight into the behaviour of software developers with regard to resolving code smells and anti-patterns. In a case study, we investigate the lifespan of code smells and the refactoring behaviour of developers in seven open source systems. The results of this study indicate that engineers are aware of code smells, but are not very concerned with their impact, given the low refactoring activity.**

I. Introduction

Software evolution can be loosely defined as the study and management of the process of repeatedly making changes to software over time for various reasons [1]. In this context Lehman [1] has observed that change is inevitable if a software system wants to remain successful. Furthermore, the *successful* evolution of software is becoming increasingly critical, given the growing dependence on software at all levels of society and economy [2].

Unfortunately, changes to a software system sometimes introduce inconsistencies in its design, thereby invalidating the original design [2] and causing the structure of the software to degrade. This structural degradation makes subsequent software evolution harder, thereby standing in the way of a successful software product.

While many types of inconsistencies can possibly be introduced into the design of a system (e.g., unforeseen exception cases and conflicting naming conventions), this study focuses on a particular type of inconsistency called an anti-pattern. An *anti-pattern* is defined by Brown et al. [3] as a commonly occurring solution that will always generate negative consequences when it is applied to a recurring problem. Detection of anti-patterns typically happens through code smells, which are symptoms of anti-patterns [4]. Examples include god classes, large methods, long parameter lists and code duplication [5].

In this study we investigate the lifespan of several code smells. In order to do so, we follow a software repository mining approach, i.e., we extract (implicit) information from version control systems about how developers work on a system [6]. In particular, for each code smell we determine when the infection takes place, i.e., when the code smell is introduced and when the underlying cause is refactored.

Having knowledge of the lifespans of code smells, and thus which code smells tend to stay in the source code for a long time, provides insight into the perspective and awareness of software developers on code smells. Our research is steered by the following research questions:

RQ1 Are some types of code smells refactored more and quicker than other smell types?
RQ2 Are relatively more code smells being refactored at an early or later stage of a system's life cycle?
RQ3 Do some developers refactor more code smells than others and to what extent?
RQ4 What refactoring rationales for code smells can be identified?

The structure of this paper is as follows: Section II provides some background, after which Section III provides details of the implementation of our tooling. Section IV presents our case study and its results. Section V discusses threats to validity, while Section VI introduces related work. Section VII concludes this paper.

II. Background

This section provides theoretical background information on the subjects related to this study.

A. Code Smells

There is no widely accepted definition of code smells. In the introduction, we described code smells as symptoms of a deeper problem, also known as an anti-pattern. In fact, code smells can be considered anti-patterns at programming level rather than design level. Smells such as large classes and methods, poor information hiding and redundant message passing are regarded as bad practices by many software engineers. However, there is some subjectivity to this determination. What developer A sees as a code smell may be considered by developer B as a valuable solution with acceptable negative side effects. Naturally, this also depends on the context, the programming language and the development methodology.

The interpretation most widely used in literature is the one by Fowler [5]. He sees a code smell as a structure that needs

Developers are aware of code smells,
but not very concerned about their impact
[Peters and Zaidman - CSMR 2012]

Studies tried explaining their lifespan

Innovations Syst Softw Eng
DOI 10.1007/s11334-013-0205-z

SI: QUATIC 2010

Investigating the evolution of code smells in object-oriented systems

Alexander Chatzigeorgiou · Anastasios Manakos

Received: 29 June 2011 / Accepted: 6 April 2013

Abstract Software design problems are known and perceived under many different terms, such as code smells, flaws, non-compliance to design principles, violation of heuristics, excessive metric values and anti-patterns, signifying the importance of handling them in the construction and maintenance of software. Once a design problem is identified, it can be removed by applying an appropriate refactoring, improving in most cases several aspects of quality such as maintainability, comprehensibility and reusability. This paper, taking advantage of recent advances and tools in the identification of non-trivial code smells, explores the presence and evolution of such problems by analyzing past versions of code. Several interesting questions can be investigated such as whether the number of problems increases with the passage of software generations, whether problems vanish by time or only by targeted human intervention, whether code smells occur in the course of evolution of a module or exist right from the beginning and whether refactorings targeting at smell removal are frequent. In contrast to previous studies that investigate the application of refactorings in the history of a software project, we attempt to analyze the evolution from the point of view of the problems themselves. To this end, we classify smell evolution patterns distinguishing deliberate maintenance activities from the removal of design problems as a side effect of software evolution. Results are discussed for two open-source systems and four code smells.

Keywords Code smell · Refactoring · Software repositories · Software history · Evolution

A. Chatzigeorgiou (✉) · A. Manakos
Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece
e-mail: achat@uom.gr

A. Manakos
e-mail: mai0932@uom.gr

1 Introduction

The design of software systems can exhibit several problems which can be either due to inefficient analysis and design during the initial construction of the software or more often, due to software ageing, where software quality degenerates over time [27]. Declining quality of evolving systems is also something that is expected according to Lehman's 7th law of software evolution [18]. The importance that the software engineering community places on the detection and resolution of design problems is evident from the multitude of terms under which they are known. Some researchers view problems as non-compliance with design principles [20], violations of design heuristics [29], excessive metric values, lack of design patterns [12] or even application of anti-patterns [3].

According to Fowler [11], design problems appear as "bad smells" at code or design level and the process of removing them consists in the application of an appropriate refactoring, i.e. an improvement in software structure without any modification of its behavior. Refactorings have been widely acknowledged mainly because of their simplicity which allows the automation of their application. Moreover, despite their simplicity, the cumulative effect of successive refactorings on design quality can be significant. Their popularity is also evident from the availability of numerous tools that provide support for the application of refactorings relieving the designers from the burden of their mechanics [24].

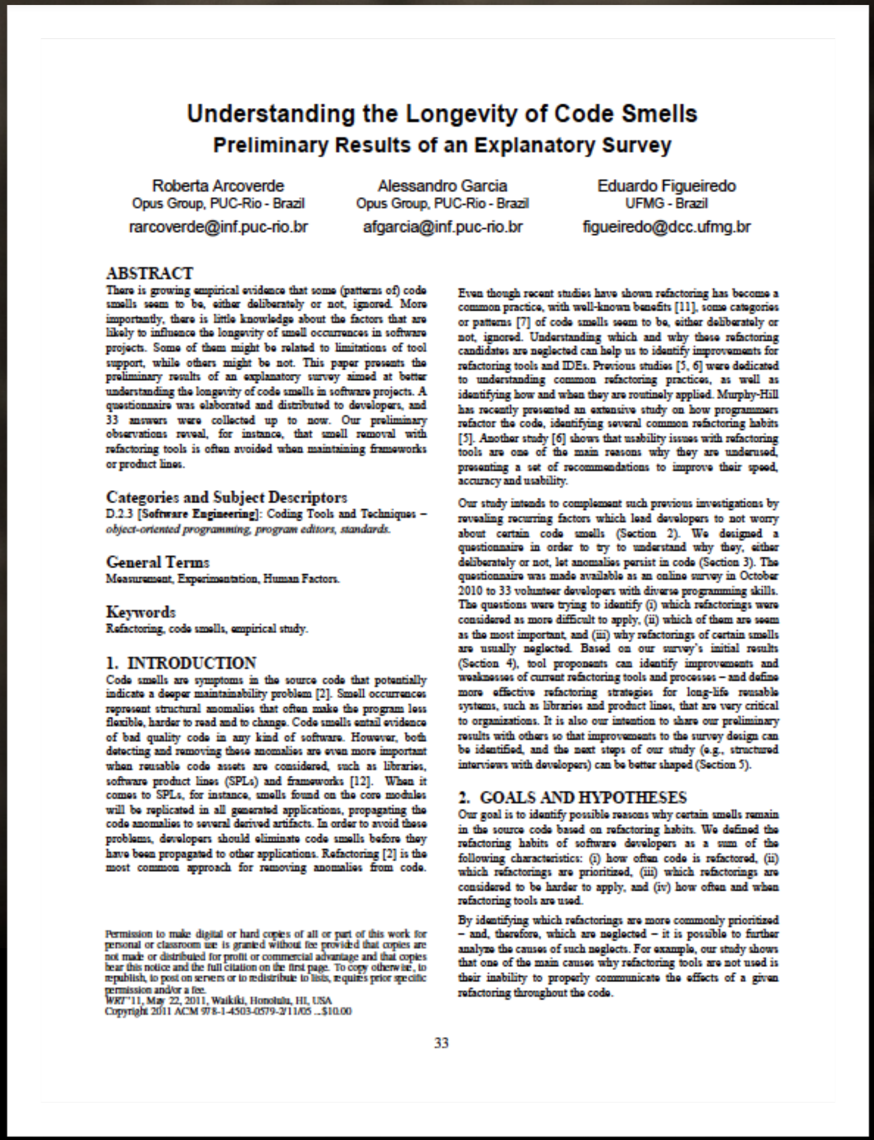
According to the recommendations proposed by Lehman and Ramil for software evolution planning [18], quality should be continuously monitored as systems evolve. This implies that past versions of a software system should be analyzed to track changes in evolutionary trends. To this end, organized collections of software repositories offer an addi-

Published online: 21 April 2013

In the vast majority of these cases code smell disappearance was not the result of targeted refactoring activities but rather a side effect of adaptive maintenance.

[Chatzigeorgiou et al. - QUATIC 2010]

… their evolution



Understanding the Longevity of Code Smells
Preliminary Results of an Explanatory Survey

Roberta Arcoverde
Opus Group, PUC-Rio - Brazil
rarcoverde@inf.puc-rio.br

Alessandro Garcia
Opus Group, PUC-Rio - Brazil
afgarcia@inf.puc-rio.br

Eduardo Figueiredo
UFMG - Brazil
figueiredo@dcc.ufmg.br

ABSTRACT
There is growing empirical evidence that some (patterns of) code smells seem to be, either deliberately or not, ignored. More importantly, there is little knowledge about the factors that are likely to influence the longevity of smell occurrences in software projects. Some of them might be related to limitations of tool support, while others might be not. This paper presents the preliminary results of an explanatory survey aimed at better understanding the longevity of code smells in software projects. A questionnaire was elaborated and distributed to developers, and 33 answers were collected up to now. Our preliminary observations reveal, for instance, that smell removal with refactoring tools is often avoided when maintaining frameworks or product lines.

Categories and Subject Descriptors
D.2.3 [**Software Engineering**]: Coding Tools and Techniques – *object-oriented programming, program editors, standards.*

General Terms
Measurement, Experimentation, Human Factors.

Keywords
Refactoring, code smells, empirical study.

1. INTRODUCTION
Code smells are symptoms in the source code that potentially indicate a deeper maintainability problem [2]. Smell occurrences represent structural anomalies that often make the program less flexible, harder to read and to change. Code smells entail evidence of bad quality code in any kind of software. However, both detecting and removing these anomalies are even more important when reusable code assets are considered, such as libraries, software product lines (SPLs) and frameworks [12]. When it comes to SPLs, for instance, smells found on the core modules will be replicated in all generated applications, propagating the code anomalies to several derived artifacts. In order to avoid these problems, developers should eliminate code smells before they have been propagated to other applications. Refactoring [2] is the most common approach for removing anomalies from code.

Even though recent studies have shown refactoring has become a common practice, with well-known benefits [11], some categories or patterns [7] of code smells seem to be, either deliberately or not, ignored. Understanding which and why these refactoring candidates are neglected can help us to identify improvements for refactoring tools and IDEs. Previous studies [5, 6] were dedicated to understanding common refactoring practices, as well as identifying how and when they are routinely applied. Murphy-Hill has recently presented an extensive study on how programmers refactor the code, identifying several common refactoring habits [5]. Another study [6] shows that usability issues with refactoring tools are one of the main reasons why they are underused, presenting a set of recommendations to improve their speed, accuracy and usability.

Our study intends to complement such previous investigations by revealing recurring factors which lead developers to not worry about certain code smells (Section 2). We designed a questionnaire in order to try to understand why they, either deliberately or not, let anomalies persist in code (Section 3). The questionnaire was made available as an online survey in October 2010 to 33 volunteer developers with diverse programming skills. The questions were trying to identify (i) which refactorings were considered as more difficult to apply, (ii) which of them are seem as the most important, and (iii) why refactorings of certain smells are usually neglected. Based on our survey's initial results (Section 4), tool proponents can identify improvements and weaknesses of current refactoring tools and processes – and define more effective refactoring strategies for long-life reusable systems, such as libraries and product lines, that are very critical to organizations. It is also our intention to share our preliminary results with others so that improvements to the survey design can be identified, and the next steps of our study (e.g., structured interviews with developers) can be better shaped (Section 5).

2. GOALS AND HYPOTHESES
Our goal is to identify possible reasons why certain smells remain in the source code based on refactoring habits. We defined the refactoring habits of software developers as a sum of the following characteristics: (i) how often code is refactored, (ii) which refactorings are prioritized, (iii) which refactorings are considered to be harder to apply, and (iv) how often and when refactoring tools are used.

By identifying which refactorings are more commonly prioritized – and, therefore, which are neglected – it is possible to further analyze the causes of such neglects. For example, our study shows that one of the main causes why refactoring tools are not used is their inability to properly communicate the effects of a given refactoring throughout the code.

WRT'11, May 22, 2011, Waikiki, Honolulu, HI, USA

Developers deliberately postpone refactorings for different reasons

[Arcoverde et al. - IWRT 2011]

and longevity…

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 40, NO. 7, JULY 2014

An Empirical Study of Refactoring Challenges and Benefits at Microsoft

Miryung Kim, *Member, IEEE*, Thomas Zimmermann, *Member, IEEE*, and Nachiappan Nagappan, *Member, IEEE*

Abstract—It is widely believed that refactoring improves software quality and developer productivity. However, few empirical studies quantitatively assess refactoring benefits or investigate developers' perception towards these benefits. This paper presents a field study of refactoring benefits and challenges at Microsoft through three complementary study methods: a survey, semi-structured interviews with professional software engineers, and quantitative analysis of version history data. Our survey finds that the refactoring definition in practice is not confined to a rigorous definition of *semantics-preserving code transformations* and that developers perceive that refactoring involves substantial cost and risks. We also report on interviews with a designated refactoring team that has led a multi-year, centralized effort on refactoring Windows. The quantitative analysis of Windows 7 version history finds the top 5 percent of preferentially refactored modules experience higher reduction in the number of inter-module dependencies and several complexity measures but increase size more than the bottom 95 percent. This indicates that measuring the impact of refactoring requires multi-dimensional assessment.

Index Terms—Refactoring, empirical study, software evolution, component dependencies, defects, churn

1 Introduction

It is widely believed that refactoring improves software quality and developer productivity by making it easier to maintain and understand software systems [1]. Many believe that a lack of refactoring incurs technical debt to be repaid in the form of increased maintenance cost [2]. For example, eXtreme programming claims that refactoring saves development cost and advocates the rule of *refactor mercilessly* throughout the entire project life cycles [3]. On the other hand, there exists a conventional wisdom that software engineers often avoid refactoring, when they are constrained by a lack of resources (e.g., right before major software releases). Some also believe that refactoring does not provide immediate benefit unlike new features or bug fixes [4].

Recent empirical studies show contradicting evidence on the benefit of refactoring as well. Ratzinger et al. [5] found that, if the number of refactorings increases in the preceding time period, the number of defects decreases. On the other hand, Weißgerber and Diehl found that a high ratio of refactoring is often followed by an increasing ratio of bug reports [6], [7] and that incomplete or incorrect refactorings cause bugs [8]. We also found similar evidence that there exists a strong correlation between the location and timing of API-level refactorings and bug fixes [9].

These contradicting findings motivated us to conduct a field study of refactoring definition, benefits, and challenges in a large software development organization and investigate whether there is a visible benefit of refactoring a large system. In this paper, we address the following research questions: (1) What is the definition of refactoring from developers' perspectives? By refactoring, do developers indeed mean behavior-preserving code transformations that modify a program structure [1], [10]? (2) What is the developers' perception about refactoring benefits and risks, and in which contexts do developers refactor code? (3) Are there visible refactoring benefits such as reduction in the number of bugs, reduction in the average size of code changes after refactoring, and reduction in the number of component dependencies?

To answer these questions, we conducted a survey with 328 professional software engineers whose check-in comments included a keyword *"refactor*"*. From our survey participants, we also came to know about a multi-year refactoring effort on Windows. Because Windows is one of the largest, long-surviving software systems within Microsoft and a designated team led an intentional effort of system-wide refactoring, we interviewed the refactoring team of Windows. Using the version history, we then assessed the impact of refactoring on various software metrics such as defects, inter-module dependencies, size and locality of code changes, complexity, test coverage, and people and organization related metrics.

To distinguish the impact of refactoring versus regular changes, we define the degree of *preferential refactoring*—applying refactorings more frequently to a module, relative to the frequency of regular changes. For example, if a module is ranked at the fifth in terms of regular commits but ranked the third in terms of refactoring commits, the rank difference is 2. This positive number indicates that, refactoring is preferentially applied to the module relative to regular commits. We use the rank difference measure specified in Section 4.4 instead of the proportion of refactoring commits out of all

- *M. Kim is with the Department of Electrical and Computer Engineering at the University of Texas at Austin.*
- *T. Zimmermann and N. Nagappan are with Microsoft Research at Redmond.*

Manuscript received 25 Mar. 2013; revised 3 Jan. 2014; accepted 16 Mar. 2014. Date of publication 17 Apr. 2014; date of current version 18 July 2014.
Recommended for acceptance by W.F. Tichy.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2014.2318734

Developers perceive refactoring involves substantial cost and risks

[Kim et al. - TSE 2014]

Why?

WHEN AND WHY YOUR CODE STARTS TO SMELL BAD

Study Design

5

Blob

Class Data Should Be Private

Complex Class

Functional Decomposition

Spaghetti Code

smells considered from the catalogues by Fowler and Brown

Class Data Should Be Private

A class exposing its attributes, violating the information hiding principle.

Complex Class

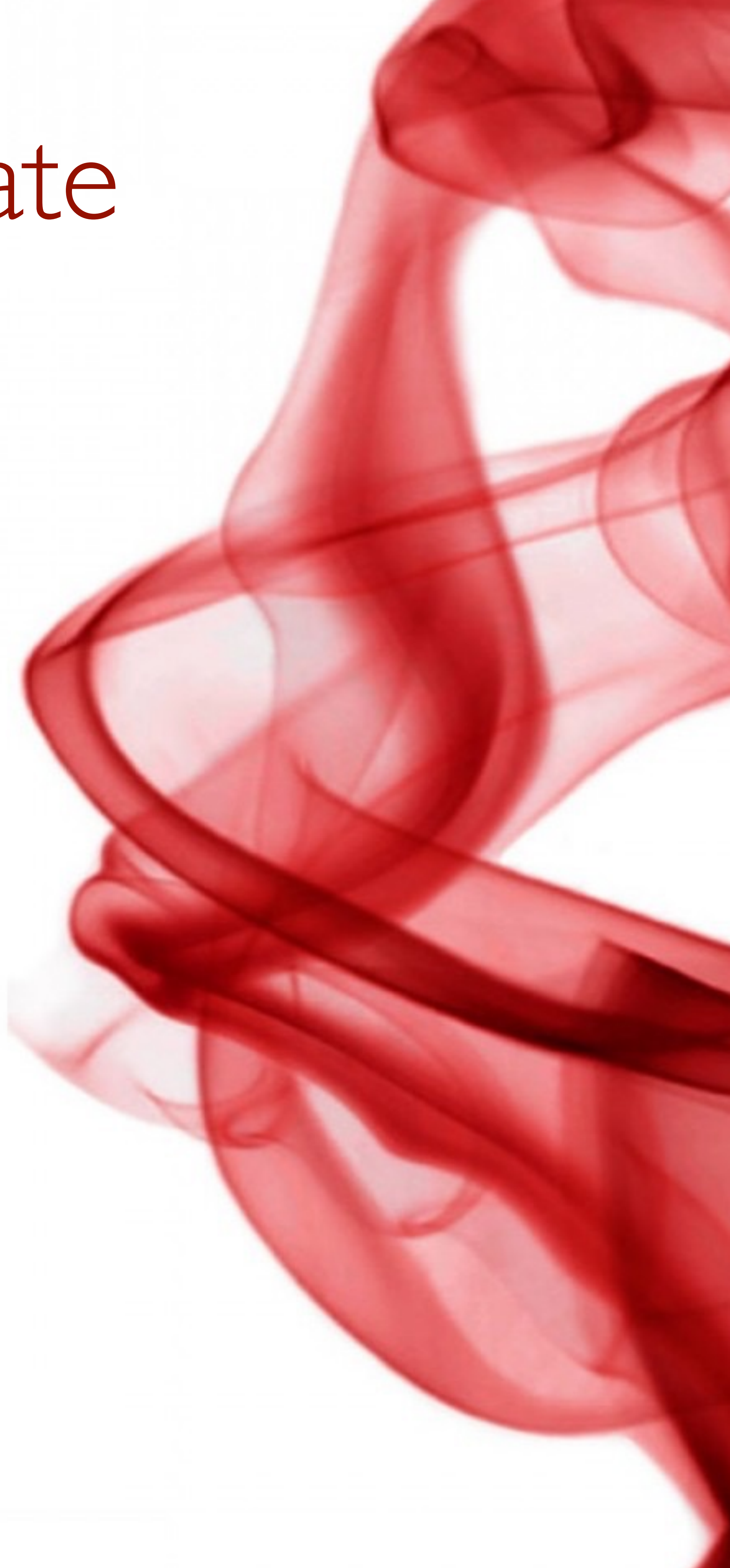
A class having high cyclomatic complexity

Functional Decomposition

A class where inheritance and polymorphism are poorly used, declaring many fields and implementing few methods

Spaghetti Code

A class without a structure that declares long methods without parameters



Study Design

3

The Apache Software Foundation

android

eclipse

different ecosystems analyzed

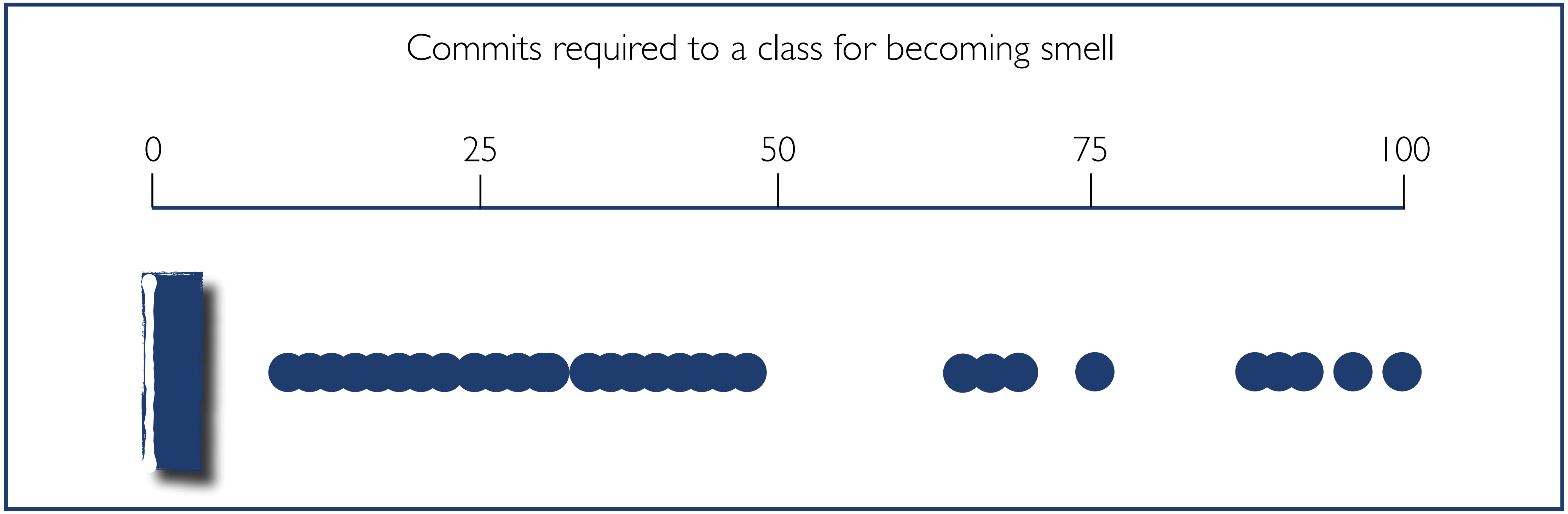
Study Design

200

total analyzed systems

When are code smells introduced

WHEN blobs are introduced

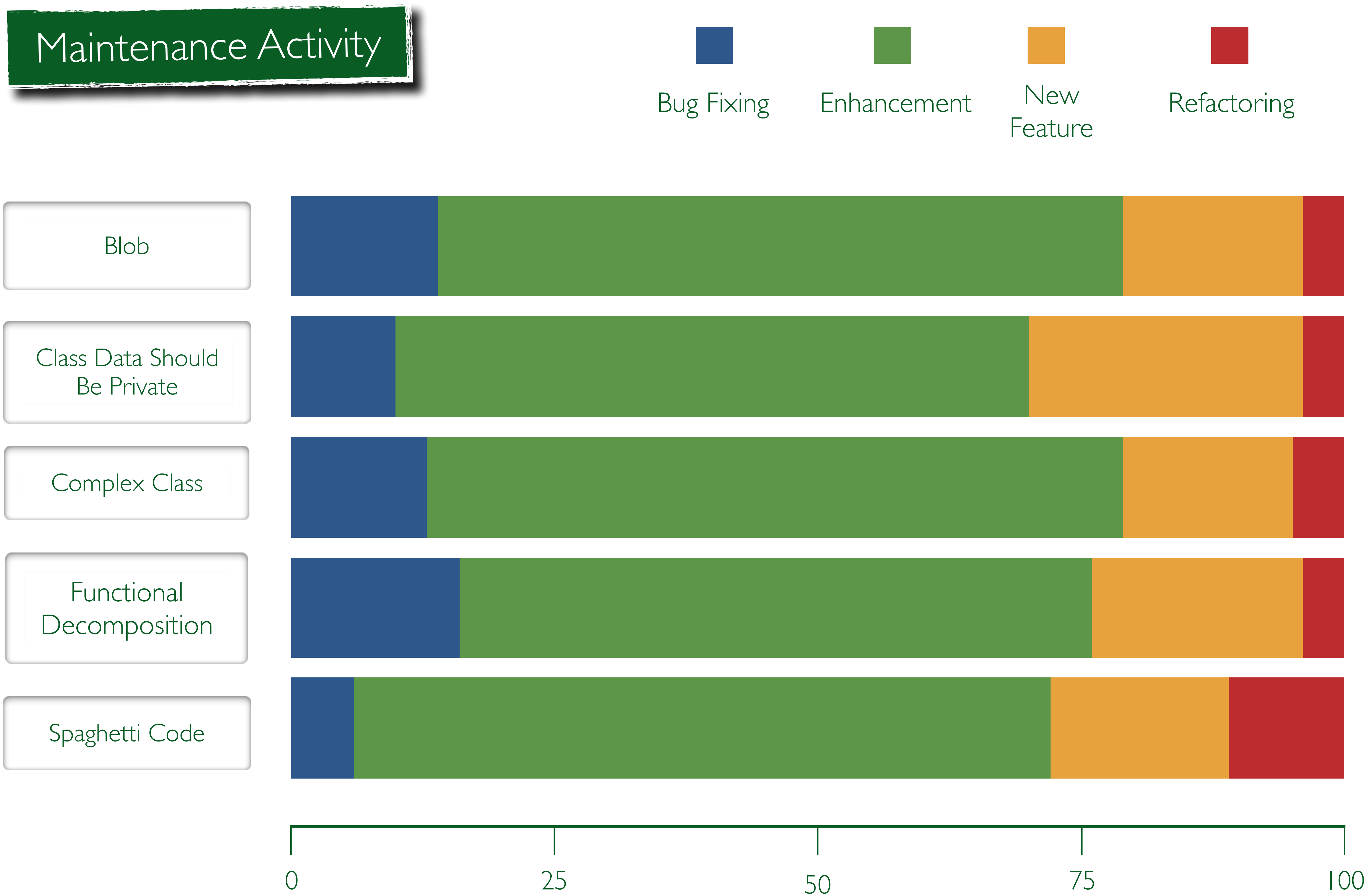


Generally, blobs affect a class since its creation

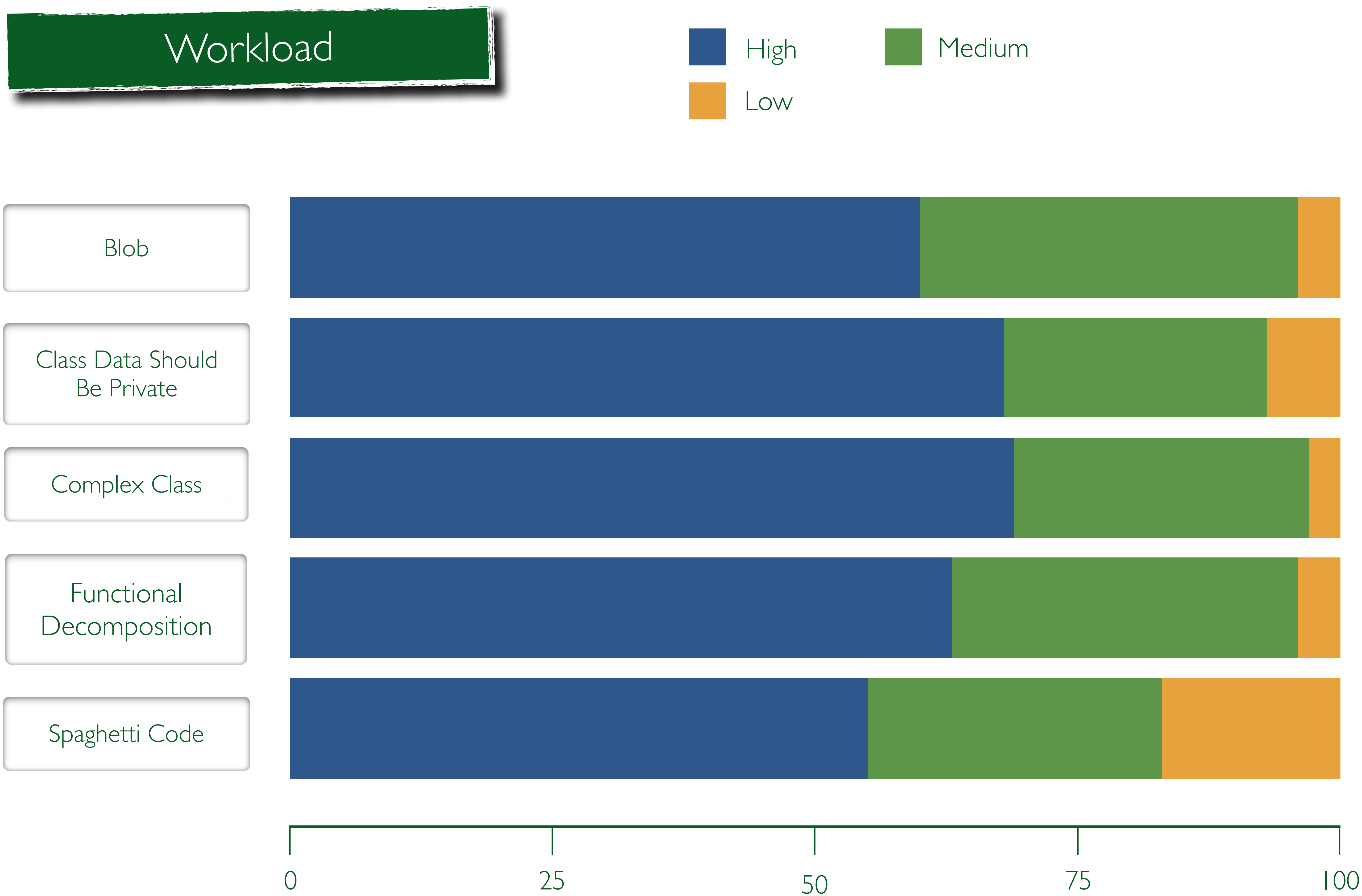
There are several cases in which a blob is introduced during maintenance activities

Why are code smells introduced

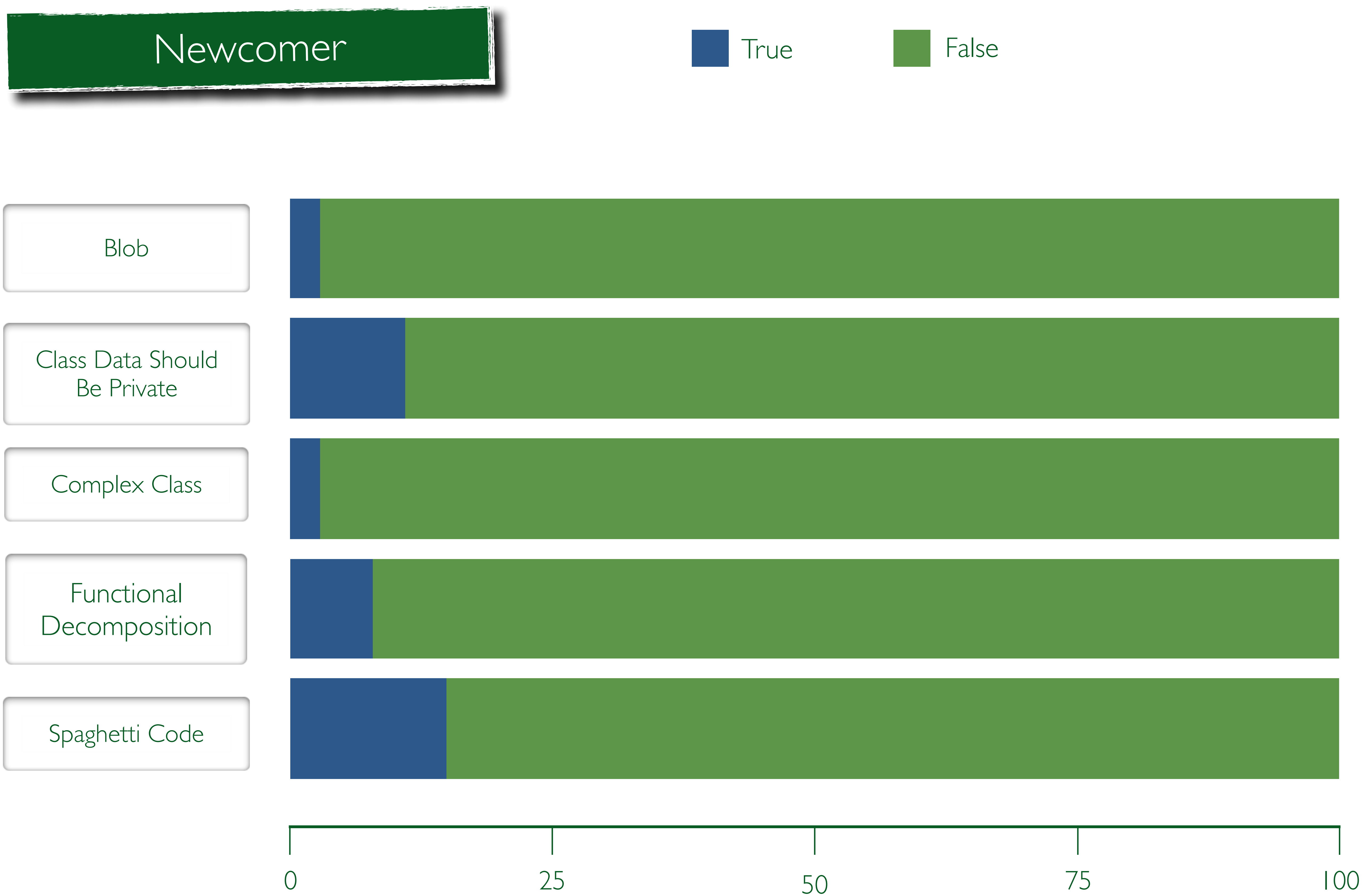
WHY are code smells introduced



WHY are code smells introduced



WHY are code smells introduced




Do They Really Smell Bad?
A Study on Developers' Perception of
Bad Code Smells


"We don't see things as they are,
we see things as we are"
Anais Nin

Study Design

Class Data Should Be Private
Complex Class
Feature Envy
God Class
Inappropriate Intimacy
Lazy Class
Long Method
Long Parameter List
Middle Man
Refused Bequest
Spaghetti Code
Speculative Generality

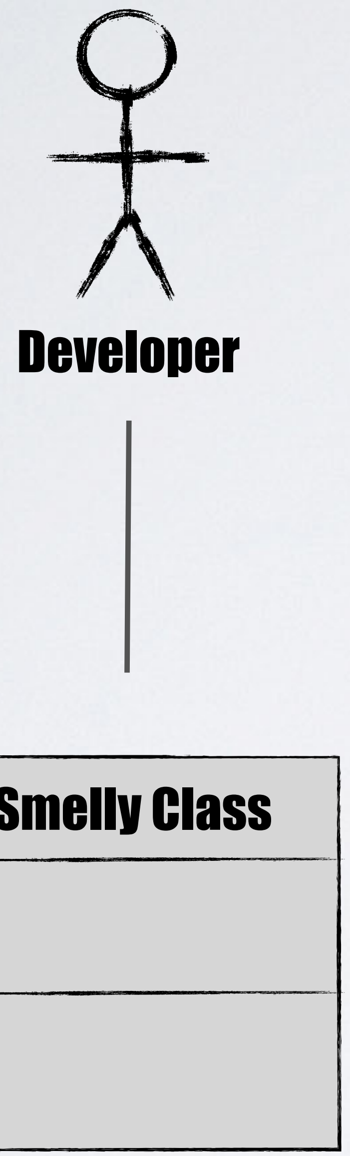
Argo UML 0.34
Eclipse 3.6.1
jEdit 4.5.1

Original Developers:
10

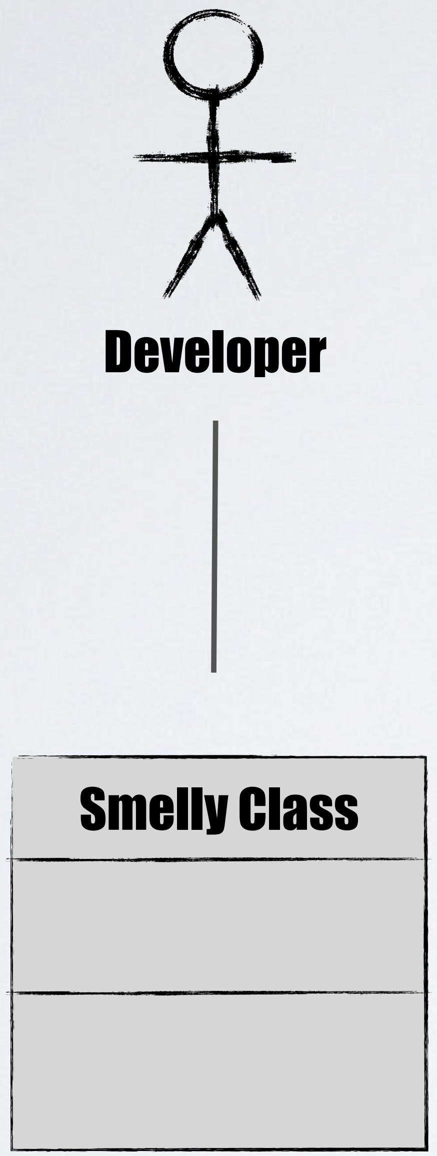
Industrial Developers
9

Master’s Students
15

Study Design

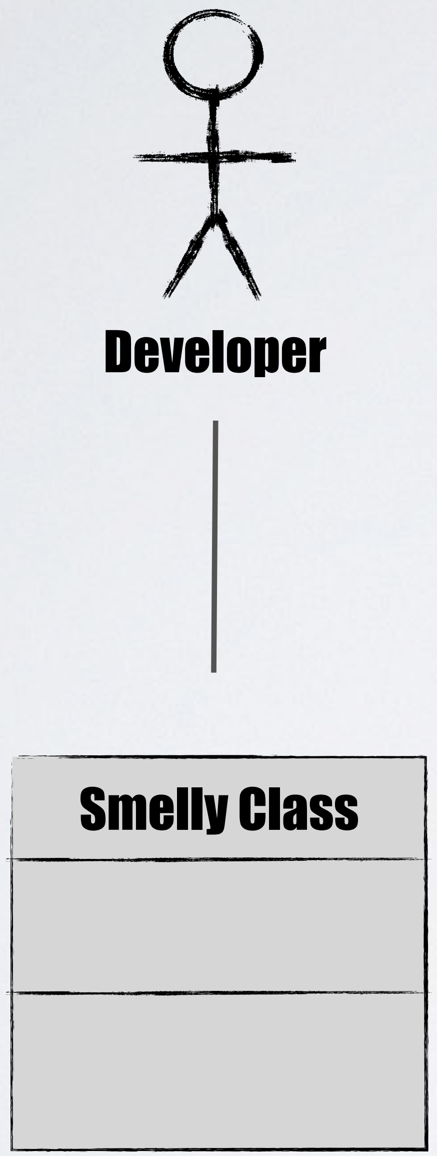


Study Design



In your opinion, does this code component exhibit any design and/or implementation problem?

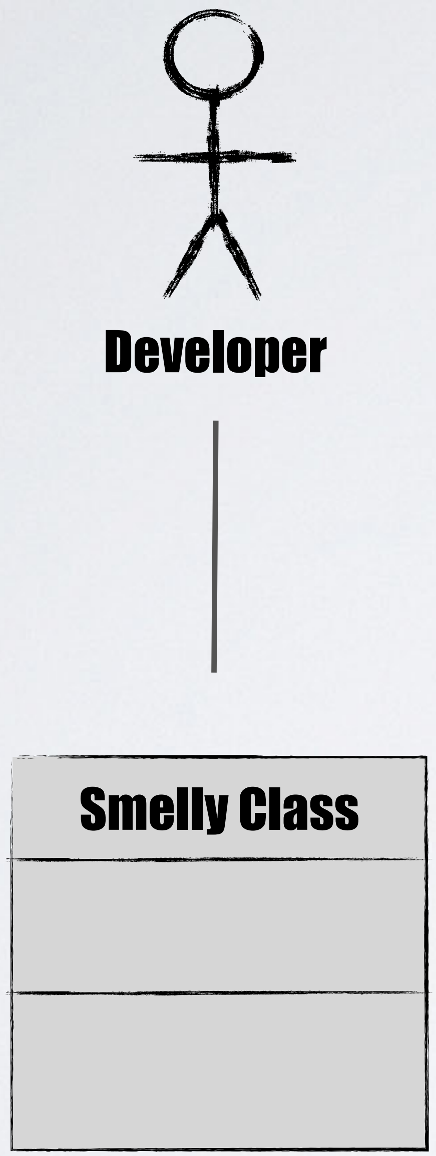
Study Design



In your opinion, does this code component exhibit any design and/or implementation problem?

- If YES, please explain what are, in your opinion, the problems affecting the code component.

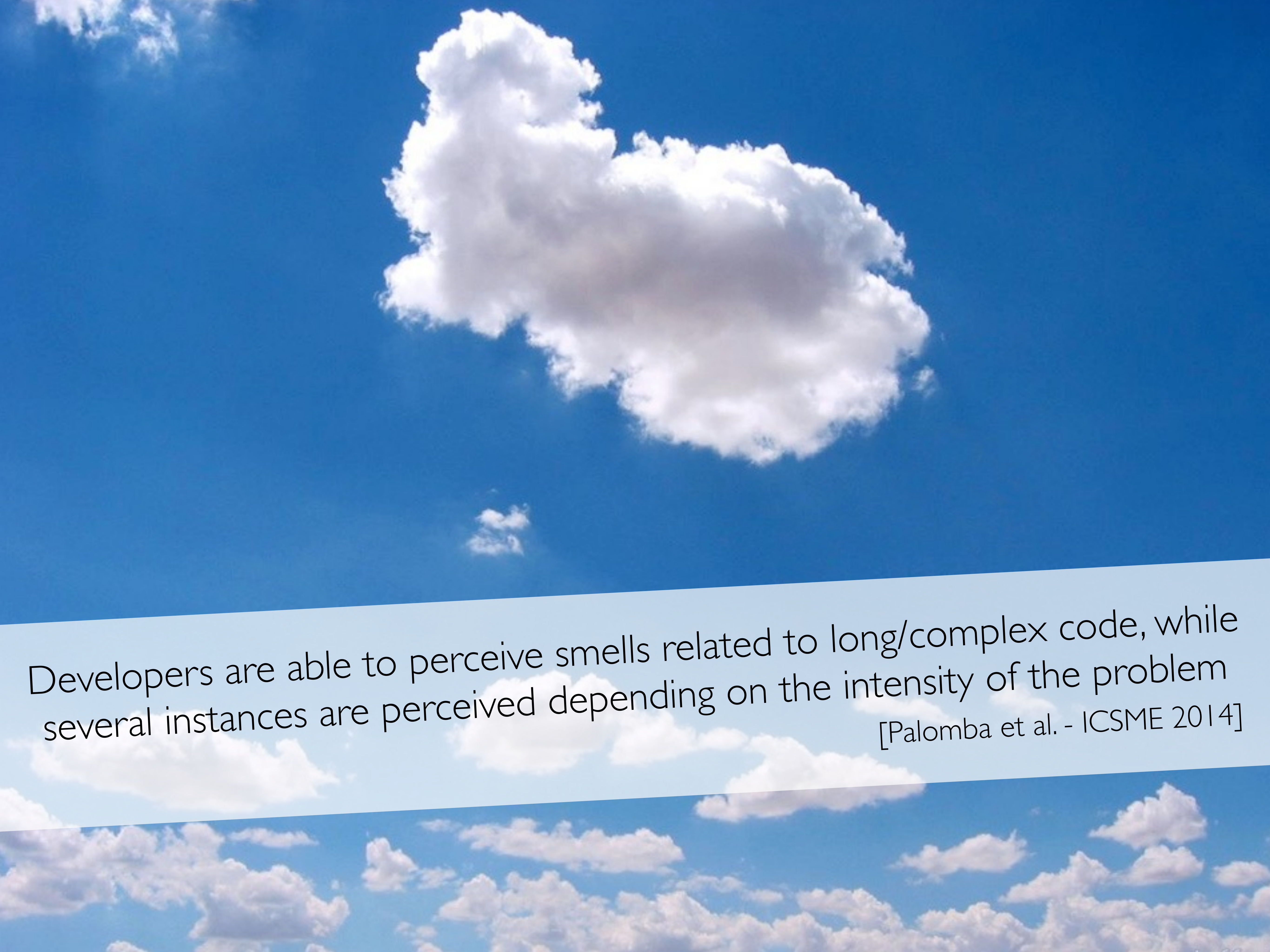
Study Design



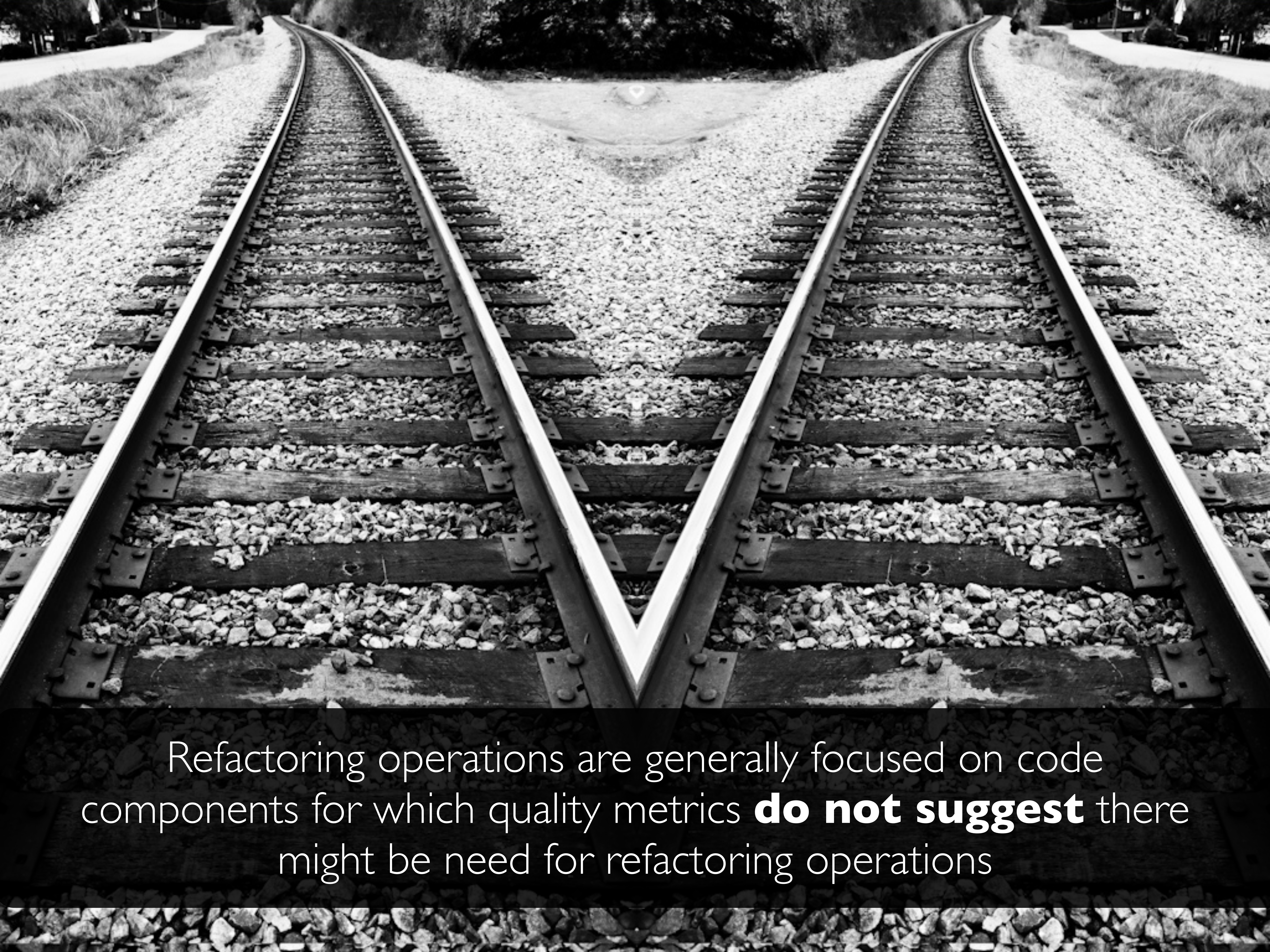
In your opinion, does this code component exhibit any design and/or implementation problem?

- If YES, please explain what are, in your opinion, the problems affecting the code component.

- If YES, please rate the severity of the design and/or implementation problem by assigning a score on the following five-points Likert scale: 1 (very low), 2 (low), 3 (medium), 4 (high), 5 (very high).


Developers are able to perceive smells related to long/complex code, while
several instances are perceived depending on the intensity of the problem
[Palomba et al. - ICSME 2014]


FEATURE ENVY
Java - ant_intermediate/src/org/apache/tools/ant/taskdefs/optional/IContract.java - Eclipse - /Users/fabiopalomba/Documents/workspaceIndigo
Quick Access
Java
IContract.java
// Set the classpath that is needed for regular Javac compilation
Path baseClasspath = createClasspath();
// Create the classpath required to automatically compile the repository files
Path repositoryClasspath = ((Path) baseClasspath.clone());
repositoryClasspath.append(new Path(getProject(), repositoryDir.getAbsolutePath()));
repositoryClasspath.append(new Path(getProject(), buildDir.getAbsolutePath()));
// Create the classpath required for iContract itself
Path iContractClasspath = ((Path) baseClasspath.clone());
iContractClasspath.append(new Path(getProject(), repositoryDir.getAbsolutePath()));
iContractClasspath.append(new Path(getProject(), instrumentDir.getAbsolutePath()));
// Create a forked java process
Java iContract = (Java) getProject().createTask("java");
iContract.setTaskName(getTaskName());
Writable
Smart Insert
563 : 13


Refactoring operations are generally focused on code components for which quality metrics **do not suggest** there might be need for refactoring operations

The relation between code smells and refactoring is stronger

42%

of refactoring operations are performed on code entities affected by code smells.

However, often refactoring fails in removing code smells!

Only 7%

of the performed operations actually remove the code smells from the affected class.

More Automation is Needed!

More Automation is Needed!

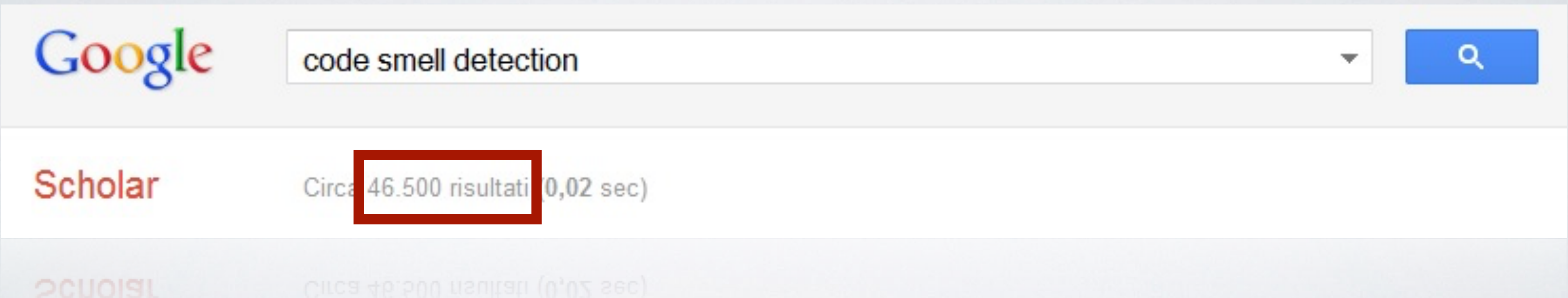
Detectors able to Take into Account the Findings on Code Smell Introduction!

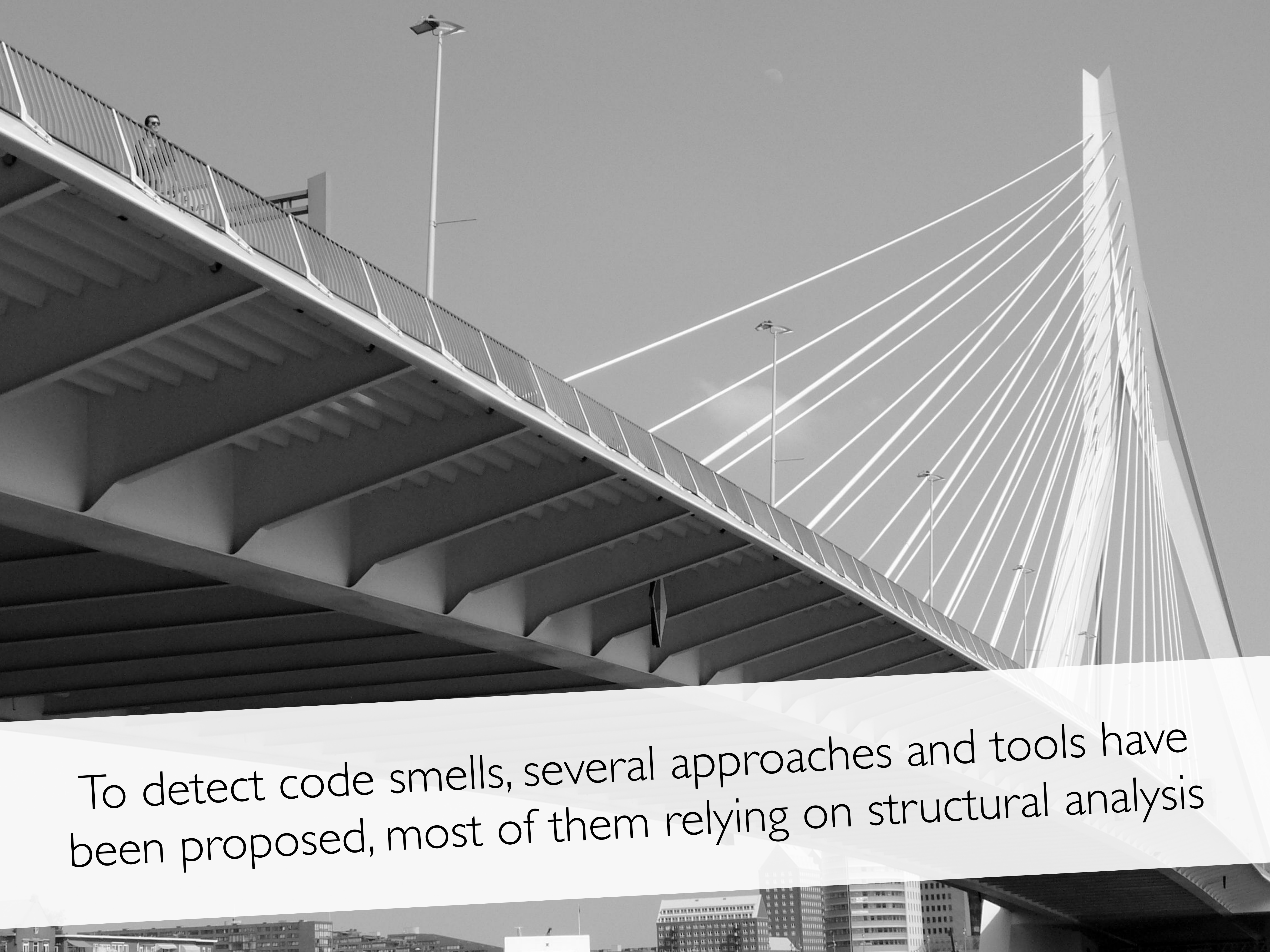
More Automation is Needed!

Detectors able to Take into Account the Findings on Code Smell Introduction!

Detectors able to Produce Suggestions Closer to the Developers' Perception of Design Problems!

Where to refactor

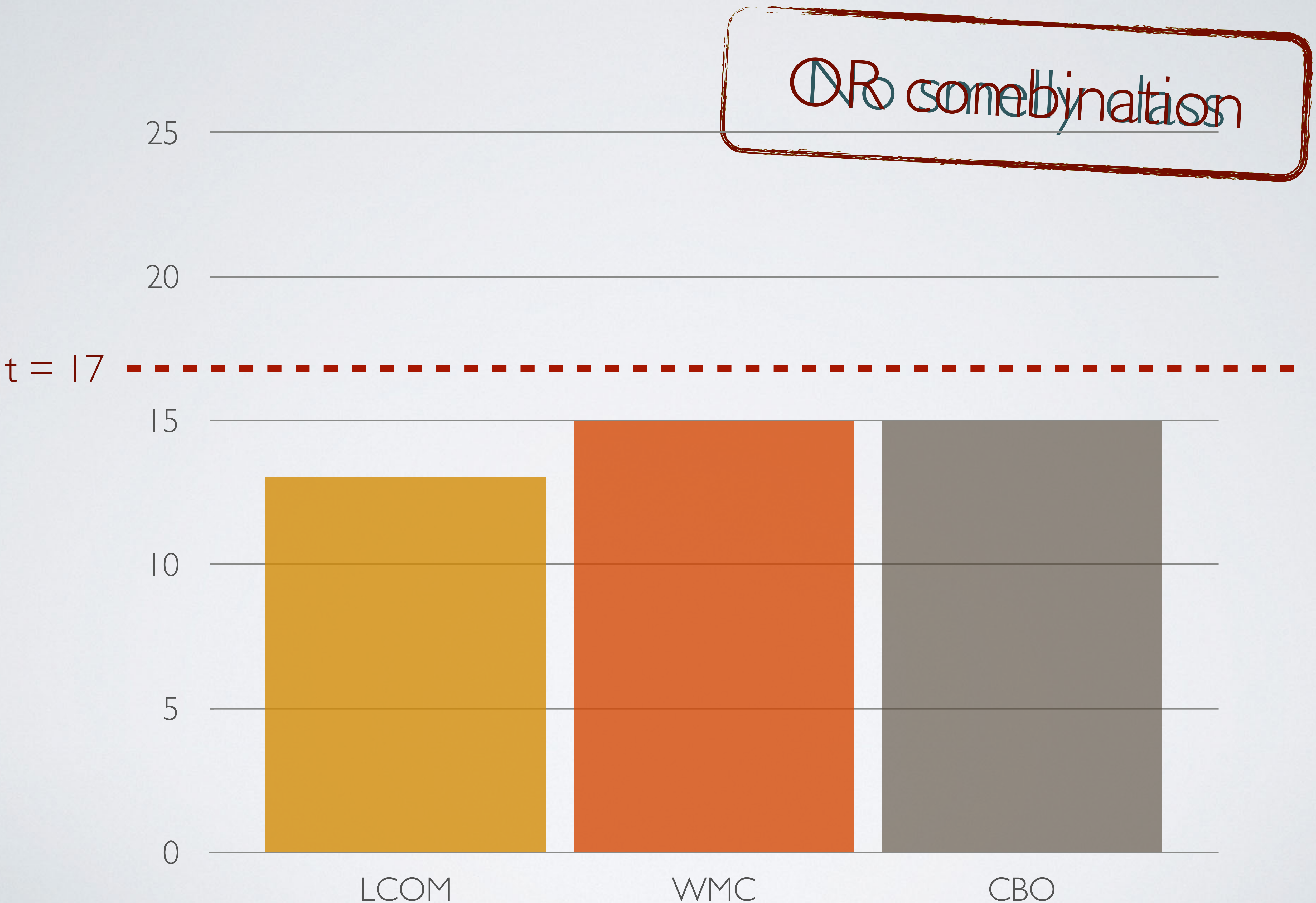




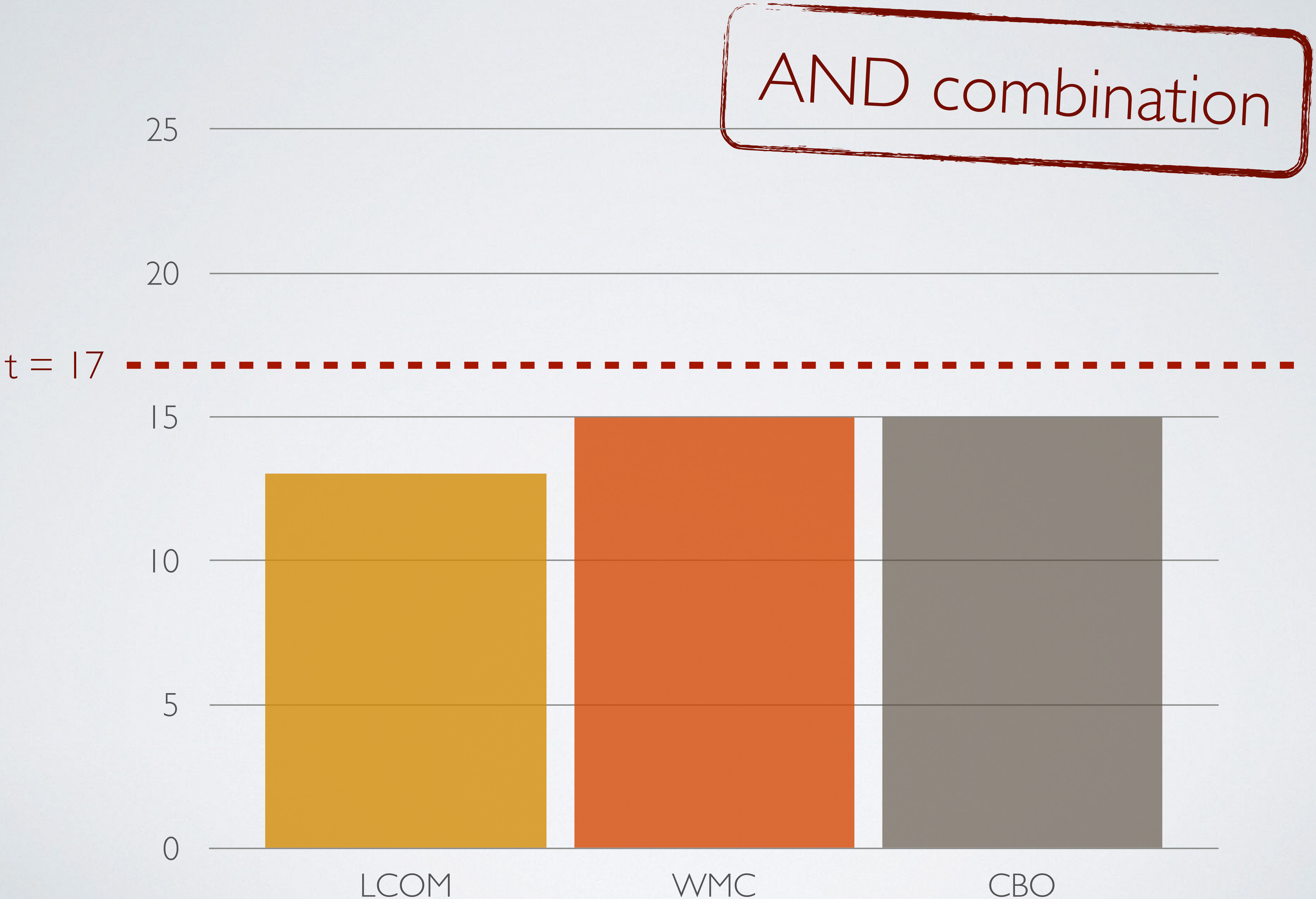
To detect code smells, several approaches and tools have been proposed, most of them relying on structural analysis


How would you detect code smells?

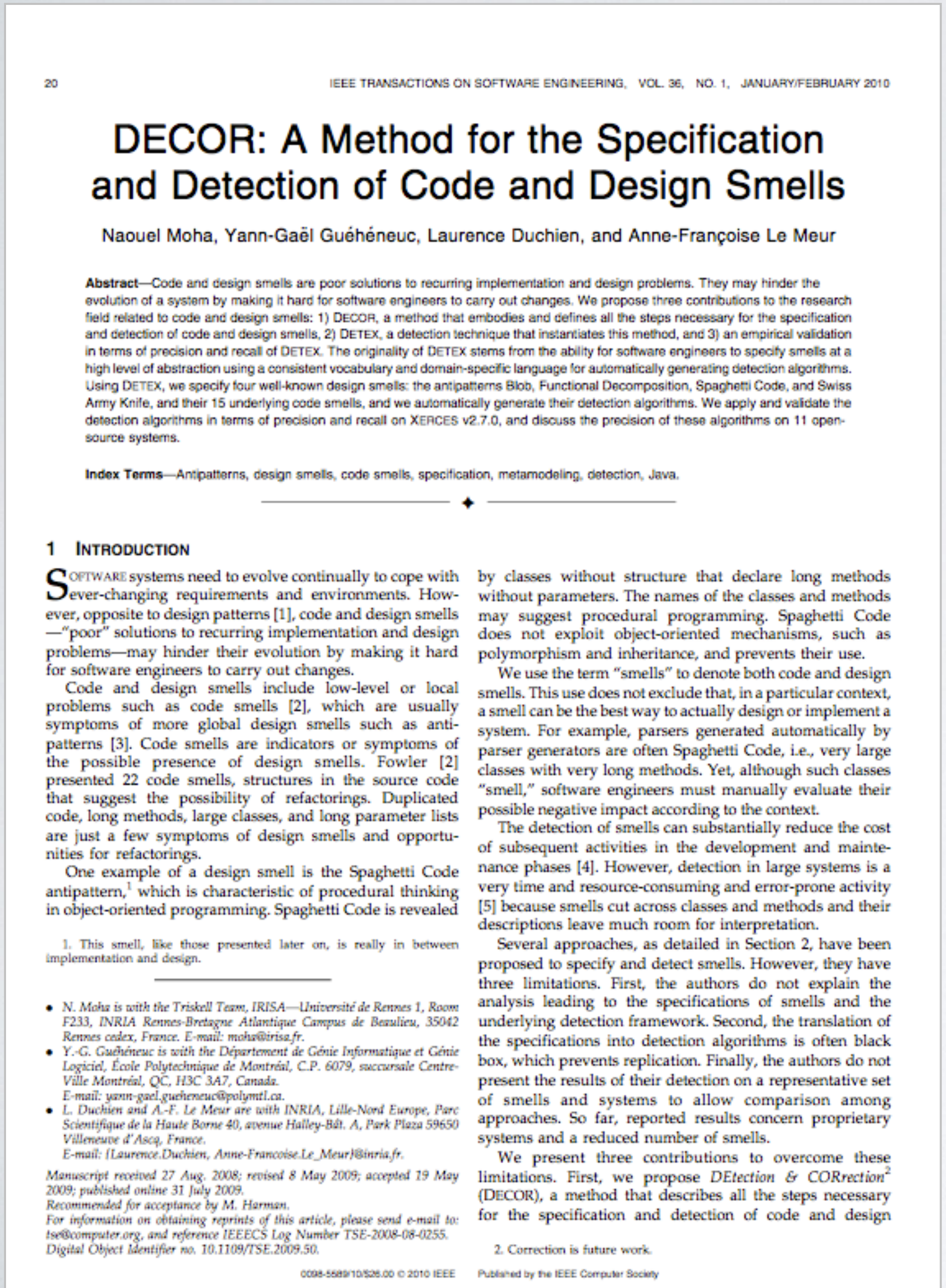
Metric-based code smell detection



Metric-based code smell detection



DECOR



DECOR: A Method for the Specification and Detection of Code and Design Smells

Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur

Abstract—Code and design smells are poor solutions to recurring implementation and design problems. They may hinder the evolution of a system by making it hard for software engineers to carry out changes. We propose three contributions to the research field related to code and design smells: 1) DECOR, a method that embodies and defines all the steps necessary for the specification and detection of code and design smells, 2) DETEX, a detection technique that instantiates this method, and 3) an empirical validation in terms of precision and recall of DETEX. The originality of DETEX stems from the ability for software engineers to specify smells at a high level of abstraction using a consistent vocabulary and domain-specific language for automatically generating detection algorithms. Using DETEX, we specify four well-known design smells: the antipatterns Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife, and their 15 underlying code smells, and we automatically generate their detection algorithms. We apply and validate the detection algorithms in terms of precision and recall on XERCES v2.7.0, and discuss the precision of these algorithms on 11 open-source systems.

Index Terms—Antipatterns, design smells, code smells, specification, metamodeling, detection, Java.

1 Introduction

Software systems need to evolve continually to cope with ever-changing requirements and environments. However, opposite to design patterns [1], code and design smells—"poor" solutions to recurring implementation and design problems—may hinder their evolution by making it hard for software engineers to carry out changes.

Code and design smells include low-level or local problems such as code smells [2], which are usually symptoms of more global design smells such as antipatterns [3]. Code smells are indicators or symptoms of the possible presence of design smells. Fowler [2] presented 22 code smells, structures in the source code that suggest the possibility of refactorings. Duplicated code, long methods, large classes, and long parameter lists are just a few symptoms of design smells and opportunities for refactorings.

One example of a design smell is the Spaghetti Code antipattern,[1] which is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes without structure that declare long methods without parameters. The names of the classes and methods may suggest procedural programming. Spaghetti Code does not exploit object-oriented mechanisms, such as polymorphism and inheritance, and prevents their use.

We use the term "smells" to denote both code and design smells. This use does not exclude that, in a particular context, a smell can be the best way to actually design or implement a system. For example, parsers generated automatically by parser generators are often Spaghetti Code, i.e., very large classes with very long methods. Yet, although such classes "smell," software engineers must manually evaluate their possible negative impact according to the context.

The detection of smells can substantially reduce the cost of subsequent activities in the development and maintenance phases [4]. However, detection in large systems is a very time and resource-consuming and error-prone activity [5] because smells cut across classes and methods and their descriptions leave much room for interpretation.

Several approaches, as detailed in Section 2, have been proposed to specify and detect smells. However, they have three limitations. First, the authors do not explain the analysis leading to the specifications of smells and the underlying detection framework. Second, the translation of the specifications into detection algorithms is often black box, which prevents replication. Finally, the authors do not present the results of their detection on a representative set of smells and systems to allow comparison among approaches. So far, reported results concern proprietary systems and a reduced number of smells.

We present three contributions to overcome these limitations. First, we propose *DEtection & CORrection*[2] (DECOR), a method that describes all the steps necessary for the specification and detection of code and design

1. This smell, like those presented later on, is really in between implementation and design.

2. Correction is future work.

- *N. Moha is with the Triskell Team, IRISA—Université de Rennes 1, Room F233, INRIA Rennes-Bretagne Atlantique Campus de Beaulieu, 35042 Rennes cedex, France. E-mail: moha@irisa.fr.*
- *Y.-G. Guéhéneuc is with the Département de Génie Informatique et Génie Logiciel, École Polytechnique de Montréal, C.P. 6079, succursale Centre-Ville Montréal, QC, H3C 3A7, Canada. E-mail: yann-gael.gueheneuc@polymtl.ca.*
- *L. Duchien and A.-F. Le Meur are with INRIA, Lille-Nord Europe, Parc Scientifique de la Haute Borne 40, avenue Halley-Bât. A, Park Plaza 59650 Villeneuve d'Ascq, France. E-mail: {Laurence.Duchien, Anne-Francoise.Le_Meur}@inria.fr.*

Manuscript received 27 Aug. 2008; revised 8 May 2009; accepted 19 May 2009; published online 31 July 2009.
Recommended for acceptance by M. Harman.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2008-08-0255.
Digital Object Identifier no. 10.1109/TSE.2009.50.

Text-based descriptions of smells

↓

Domain Analysis

↓

Vocabulary, Taxonomy

↓

Specification

↓

Rule Cards

↓

Algorithm Generation

↓

Detection Algorithm

[Moha et al. TSE 2010]

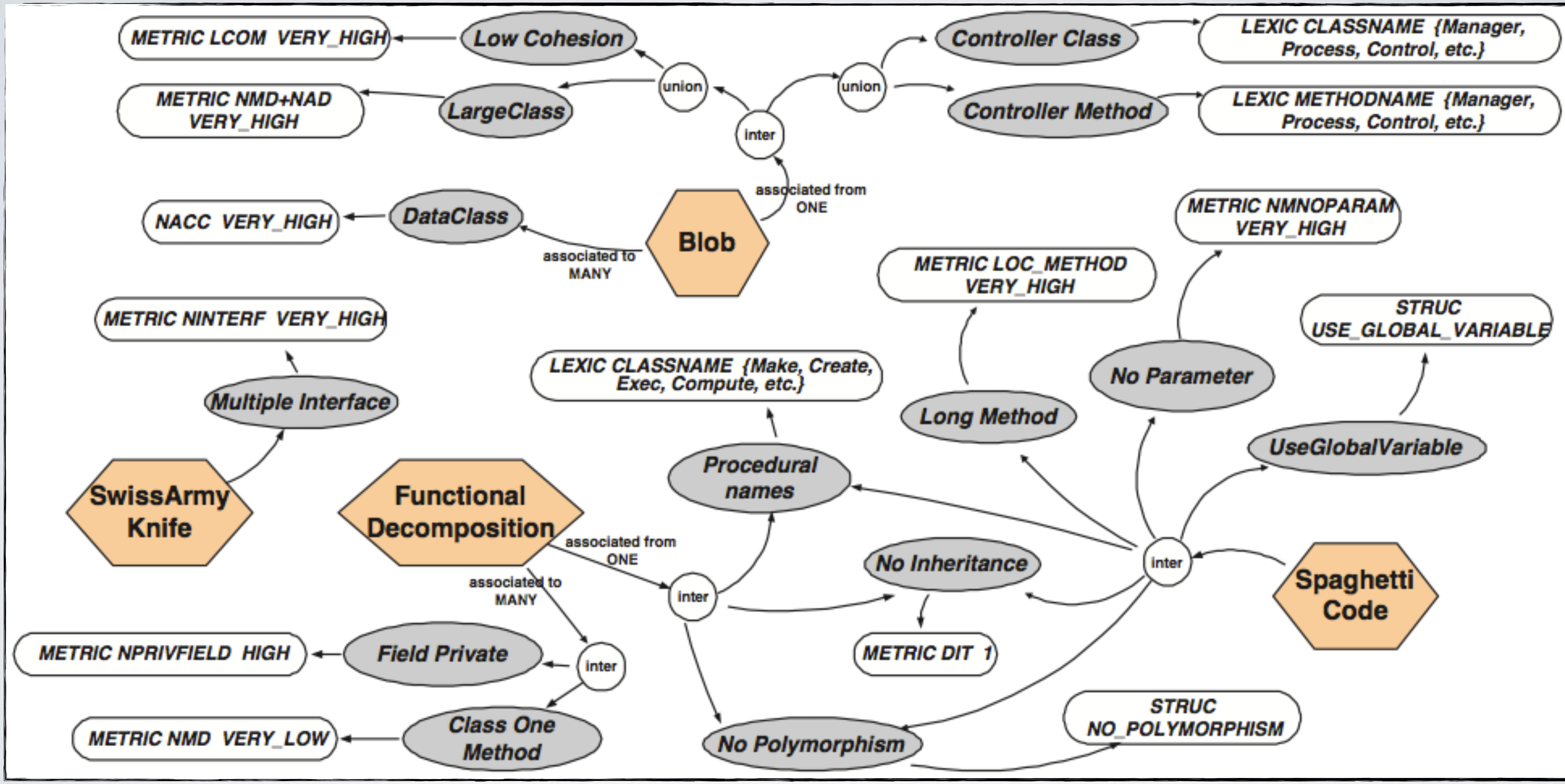
DECOR

input example

The Blob (also called God class) corresponds to a large controller class that depends on data stored in surrounding data classes. A large class declares many fields and methods with a low cohesion. A controller class monopolizes most of the processing done by a system, takes most of the decisions, and closely directs the processing of other classes. Controller classes can be identified using suspicious names such as Process, Control, Manage, System, and so on. A data class contains only data and performs no processing on these data. It is composed of highly cohesive fields and accessors.

[Moha et al. TSE 2010]

DECOR



[Moha et al. TSE 2010]

DECOR

RULE_CARD : Blob {

RULE : Blob {ASSOC: associated FROM : mainClass ONE TO : DataClass MANY};

RULE : MainClass {UNION LargeClass, LowCohesion, ControllerClass};

RULE : LargeClass {(METRIC : NMD + NAD,VERY_HIGH, 20) } ;

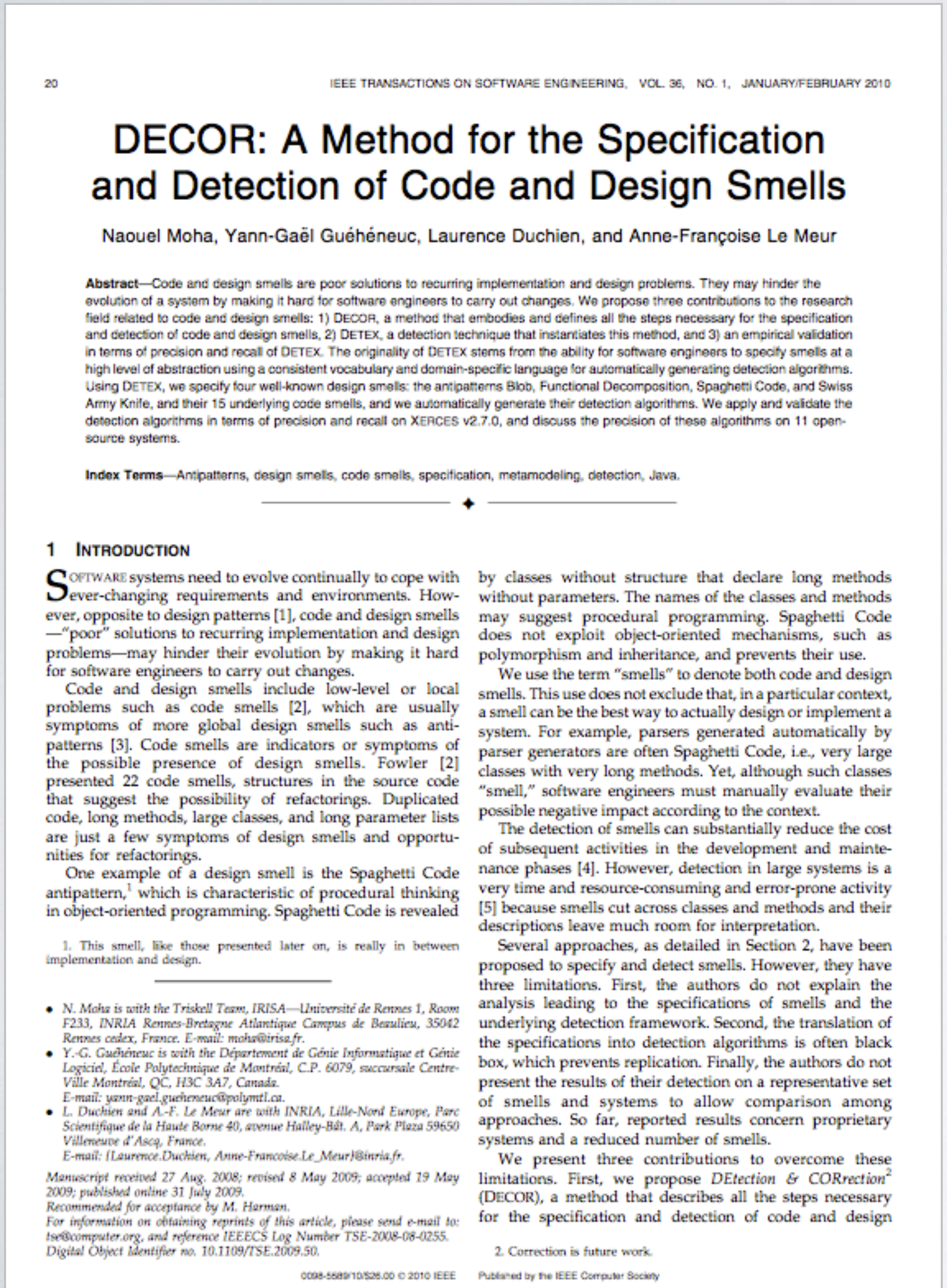
RULE : LowCohesion { (METRIC : LCOM5,VERY_HIGH , 20) } ;

RULE : ControllerClass { UNION (SEMANTIC : METHODNAME,
{Process, Control , Ctrl , Command , Cmd, Proc, UI, Manage, Drive})
(SEMANTIC : CLASSNAME, { Process, Control, Ctrl, Command , Cmd, Proc , UI,
Manage, Drive , System, Subsystem }) } ;

RULE : DataClass {(STRUCT: METHOD_ACCESSOR, 90%)} ;
};

[Moha et al. TSE 2010]

DECOR



DECOR: A Method for the Specification and Detection of Code and Design Smells

Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur

Abstract—Code and design smells are poor solutions to recurring implementation and design problems. They may hinder the evolution of a system by making it hard for software engineers to carry out changes. We propose three contributions to the research field related to code and design smells: 1) DECOR, a method that embodies and defines all the steps necessary for the specification and detection of code and design smells, 2) DETEX, a detection technique that instantiates this method, and 3) an empirical validation in terms of precision and recall of DETEX. The originality of DETEX stems from the ability for software engineers to specify smells at a high level of abstraction using a consistent vocabulary and domain-specific language for automatically generating detection algorithms. Using DETEX, we specify four well-known design smells: the antipatterns Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife, and their 15 underlying code smells, and we automatically generate their detection algorithms. We apply and validate the detection algorithms in terms of precision and recall on XERCES v2.7.0, and discuss the precision of these algorithms on 11 open-source systems.

Index Terms—Antipatterns, design smells, code smells, specification, metamodeling, detection, Java.

1 Introduction

Software systems need to evolve continually to cope with ever-changing requirements and environments. However, opposite to design patterns [1], code and design smells —"poor" solutions to recurring implementation and design problems—may hinder their evolution by making it hard for software engineers to carry out changes.

Code and design smells include low-level or local problems such as code smells [2], which are usually symptoms of more global design smells such as anti-patterns [3]. Code smells are indicators or symptoms of the possible presence of design smells. Fowler [2] presented 22 code smells, structures in the source code that suggest the possibility of refactorings. Duplicated code, long methods, large classes, and long parameter lists are just a few symptoms of design smells and opportunities for refactorings.

One example of a design smell is the Spaghetti Code antipattern,[1] which is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes without structure that declare long methods without parameters. The names of the classes and methods may suggest procedural programming. Spaghetti Code does not exploit object-oriented mechanisms, such as polymorphism and inheritance, and prevents their use.

We use the term "smells" to denote both code and design smells. This use does not exclude that, in a particular context, a smell can be the best way to actually design or implement a system. For example, parsers generated automatically by parser generators are often Spaghetti Code, i.e., very large classes with very long methods. Yet, although such classes "smell," software engineers must manually evaluate their possible negative impact according to the context.

The detection of smells can substantially reduce the cost of subsequent activities in the development and maintenance phases [4]. However, detection in large systems is a very time and resource-consuming and error-prone activity [5] because smells cut across classes and methods and their descriptions leave much room for interpretation.

Several approaches, as detailed in Section 2, have been proposed to specify and detect smells. However, they have three limitations. First, the authors do not explain the analysis leading to the specifications of smells and the underlying detection framework. Second, the translation of the specifications into detection algorithms is often black box, which prevents replication. Finally, the authors do not present the results of their detection on a representative set of smells and systems to allow comparison among approaches. So far, reported results concern proprietary systems and a reduced number of smells.

We present three contributions to overcome these limitations. First, we propose *DEtection & CORrection*[2] (DECOR), a method that describes all the steps necessary for the specification and detection of code and design

1. This smell, like those presented later on, is really in between implementation and design.

2. Correction is future work.

• *N. Moha is with the Triskell Team, IRISA—Université de Rennes 1, Room F233, INRIA Rennes-Bretagne Atlantique Campus de Beaulieu, 35042 Rennes cedex, France. E-mail: moha@irisa.fr.*
• *Y.-G. Guéhéneuc is with the Département de Génie Informatique et Génie Logiciel, École Polytechnique de Montréal, C.P. 6079, succursale Centre-Ville Montréal, QC, H3C 3A7, Canada.*
E-mail: yann-gael.gueheneuc@polymtl.ca.
• *L. Duchien and A.-F. Le Meur are with INRIA, Lille-Nord Europe, Parc Scientifique de la Haute Borne 40, avenue Halley-Bât. A, Park Plaza 59650 Villeneuve d'Ascq, France.*
E-mail: {Laurence.Duchien, Anne-Francoise.Le_Meur}@inria.fr.

Manuscript received 27 Aug. 2008; revised 8 May 2009; accepted 19 May 2009; published online 31 July 2009.
Recommended for acceptance by M. Harman.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2008-08-0255.
Digital Object Identifier no. 10.1109/TSE.2009.50.

 Published by the IEEE Computer Society

Performances

Detect instances of four code smells (i.e., Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife) on 9 software systems

Average Recall: 100%
Average Precision: 60.5%

[Moha et al. TSE 2010]

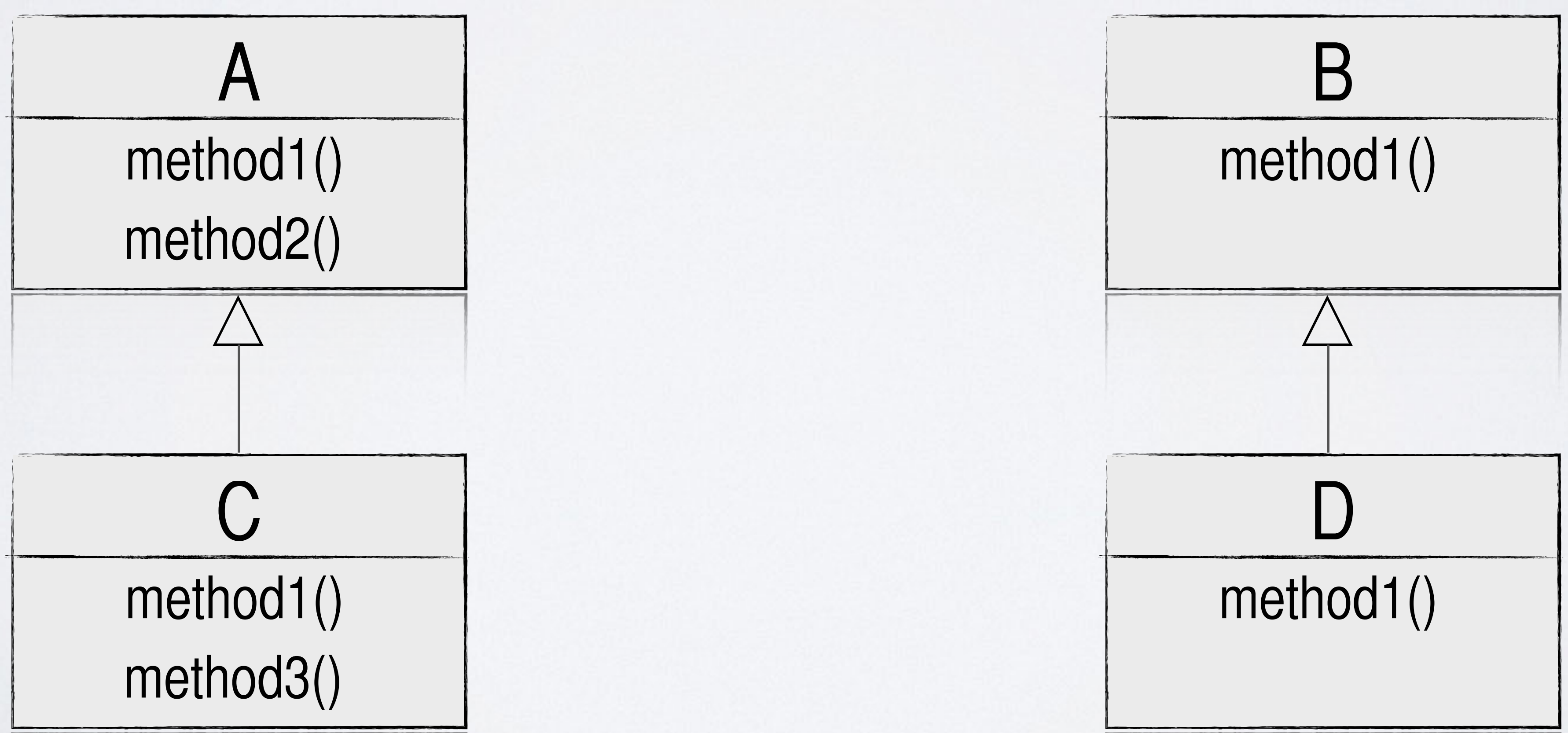
But some smells are intrinsically characterized by how code evolves over time

Parallel Inheritance

Every time you make a subclass of one class, you also have to make a subclass of another

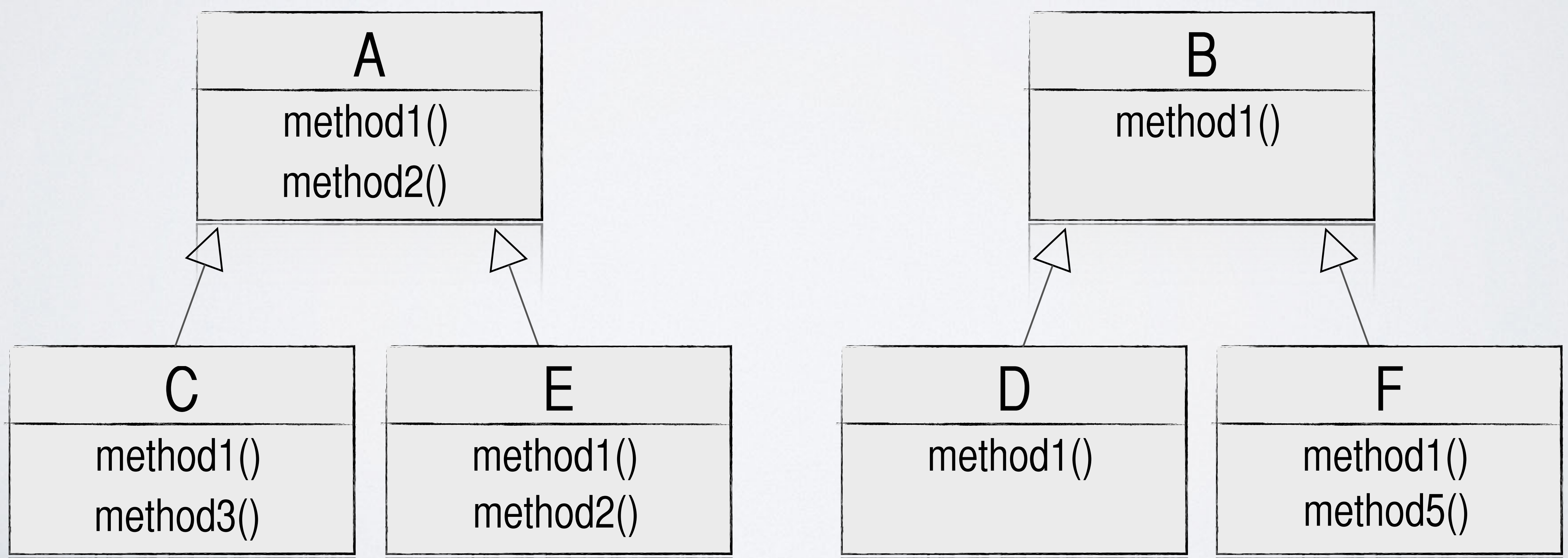
Parallel Inheritance

Every time you make a subclass of one class, you also have to make a subclass of another

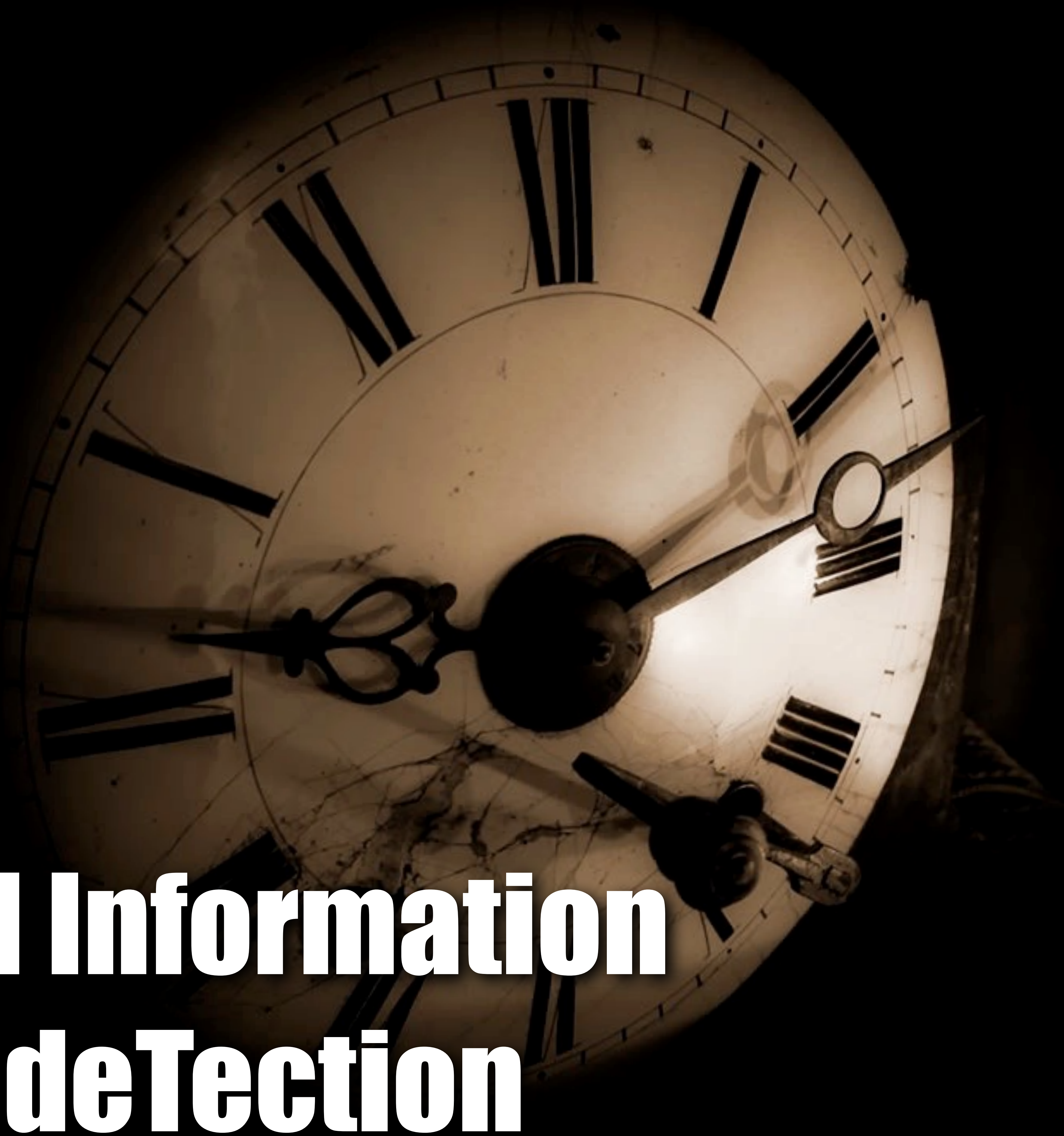


Parallel Inheritance

Every time you make a subclass of one class, you also have to make a subclass of another

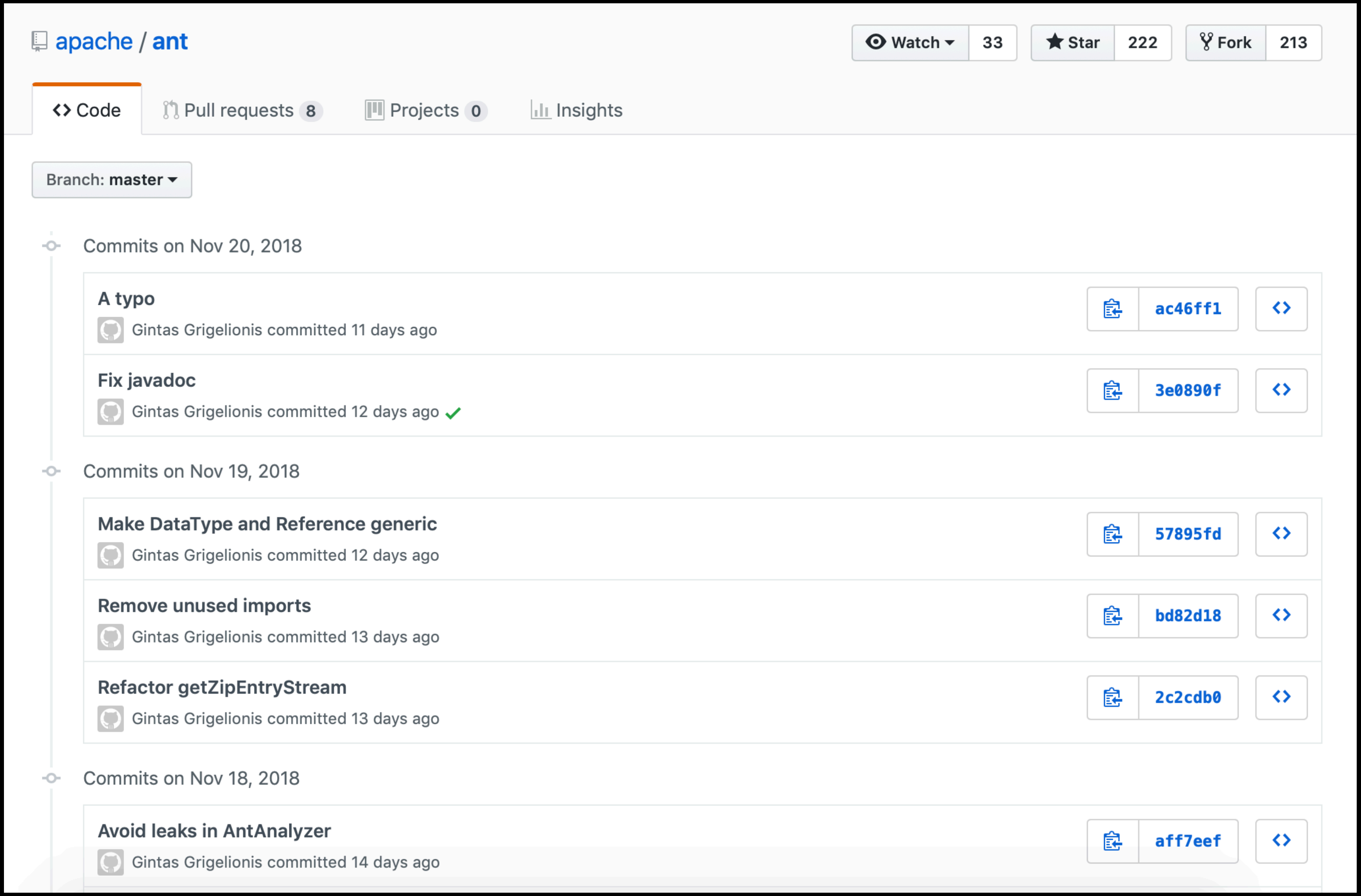


HIST

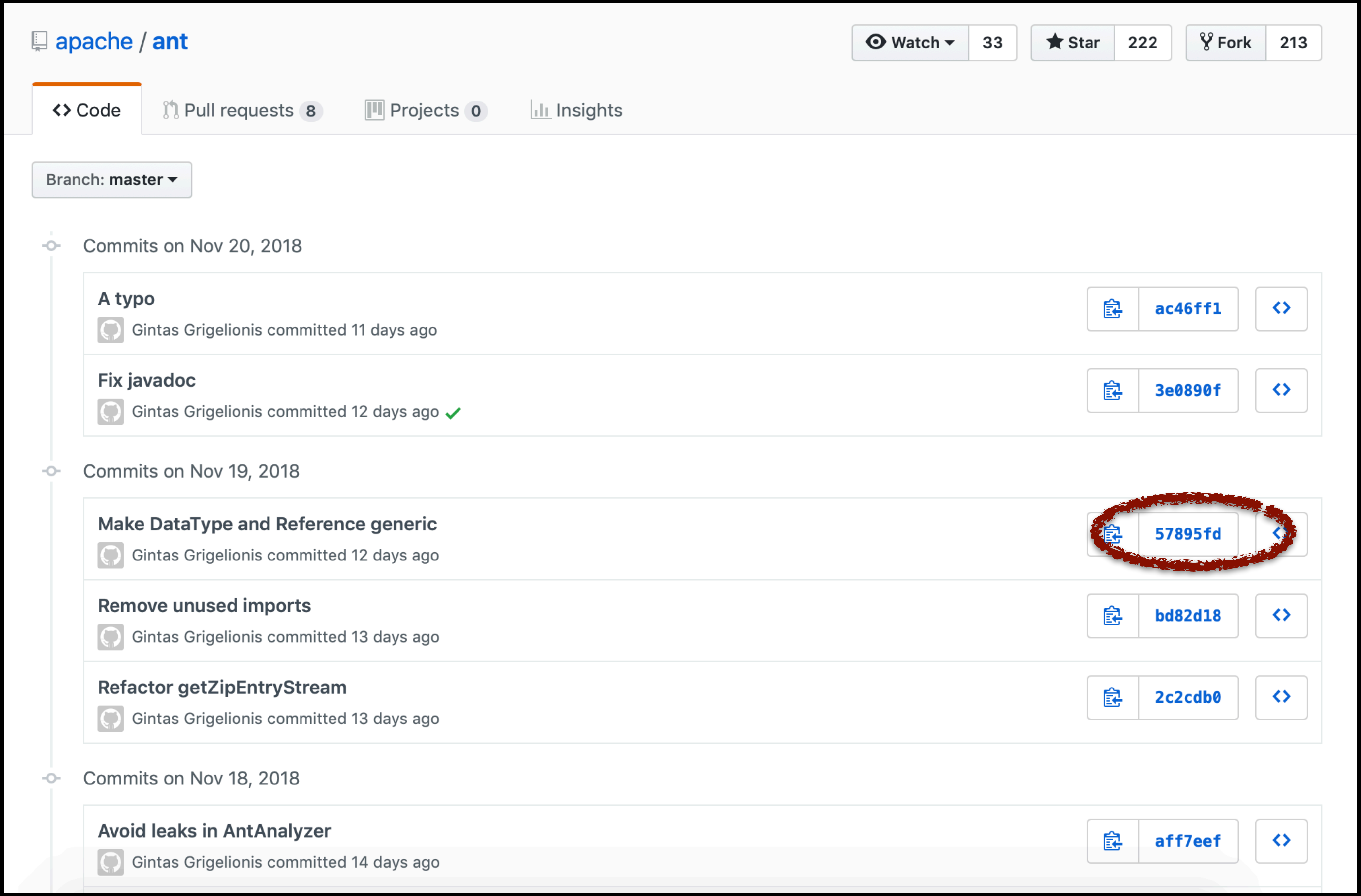


Historical Information for Smell deTection

Extracting Change History Information



Extracting Change History Information



Extracting Change History Information

Make DataType and Reference generic

master

Gintas Grigelionis committed 12 days ago   1 parent bd82d18   commit 57895fd06465933703cdb955

Showing 57 changed files with **379 additions** and **261 deletions**.

6 src/etc/testcases/taskdefs/tar.xml

```
@@ -172,6 +172,12 @@
```

172	`<untar src="${output}/test11.tar.bz2" dest="${output}/untar" compression="bzip2"/>`	172	`<untar src="${output}/test11.tar.bz2" dest="${output}/untar" compression="bzip2"/>`
173	`</target>`	173	`</target>`
174		174	
		175	`+ <target name="testTarFilesetWithReference">`
		176	`+ <fileset id="xml.fileset" dir="." includes="*.xml"/>`
		177	`+ <tar destfile="${output}/testtar.tar">`
		178	`+ <tarfileset prefix="pre" refid="xml.fileset"/>`
		179	`+ </tar>`
		180	`+ </target>`
175		181	
176	`<target name="feather">`	182	`<target name="feather">`
177	`<tar destfile="${output}/asf-logo.gif.tar"`	183	`<tar destfile="${output}/asf-logo.gif.tar"`

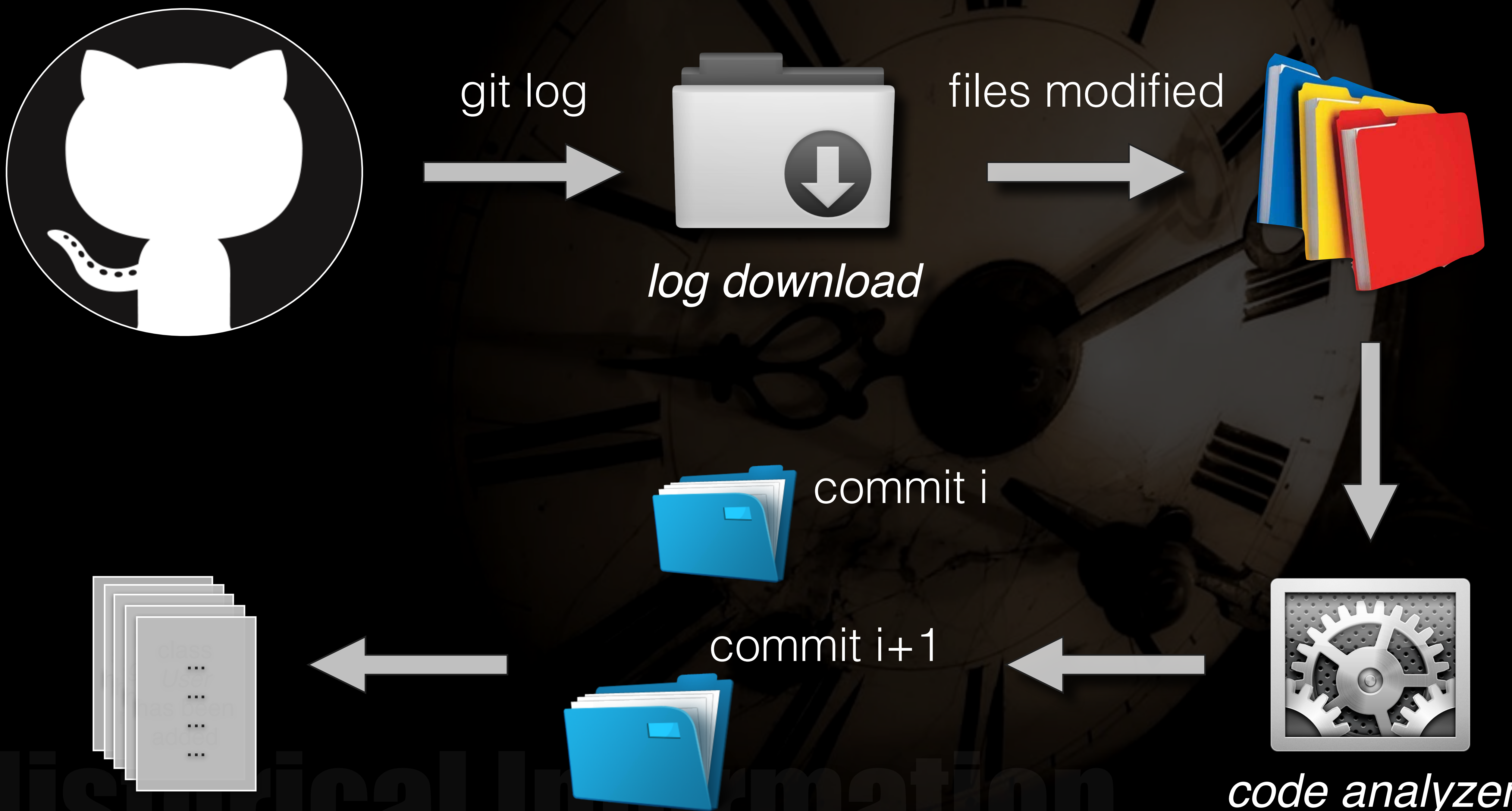
Extracting Change History Information

Historical Information

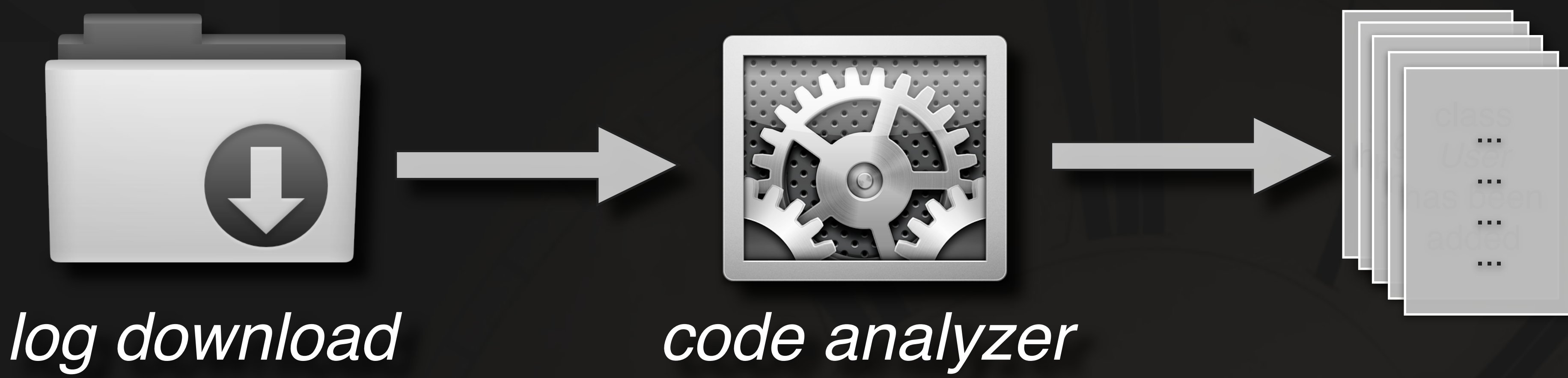
Extracting Change History Information

Historical Information

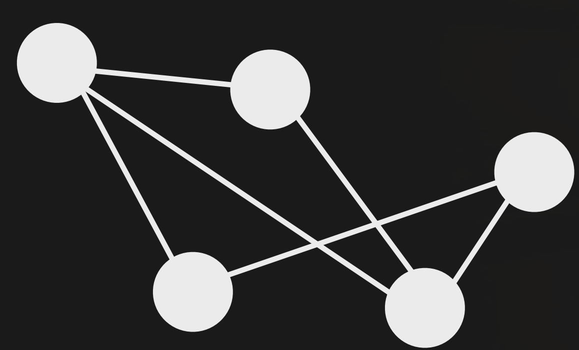
Extracting Change History Information



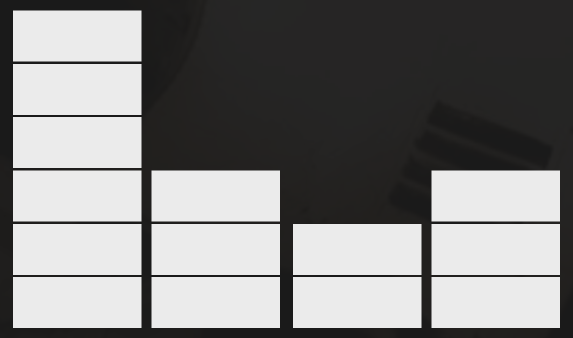
Change History Extractor



Code Smells Detector

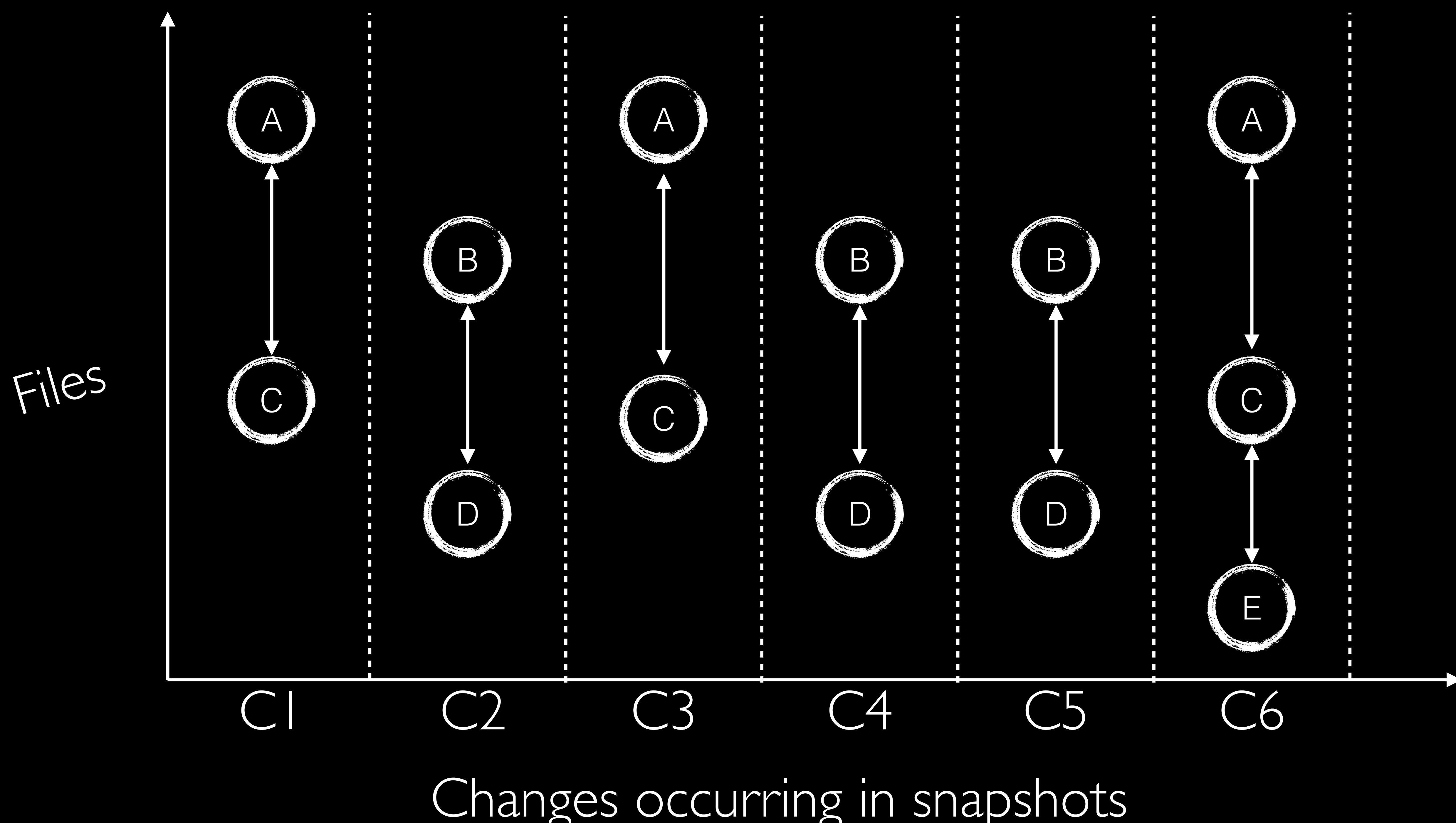


Association rule discovery to capture co-changes between entities

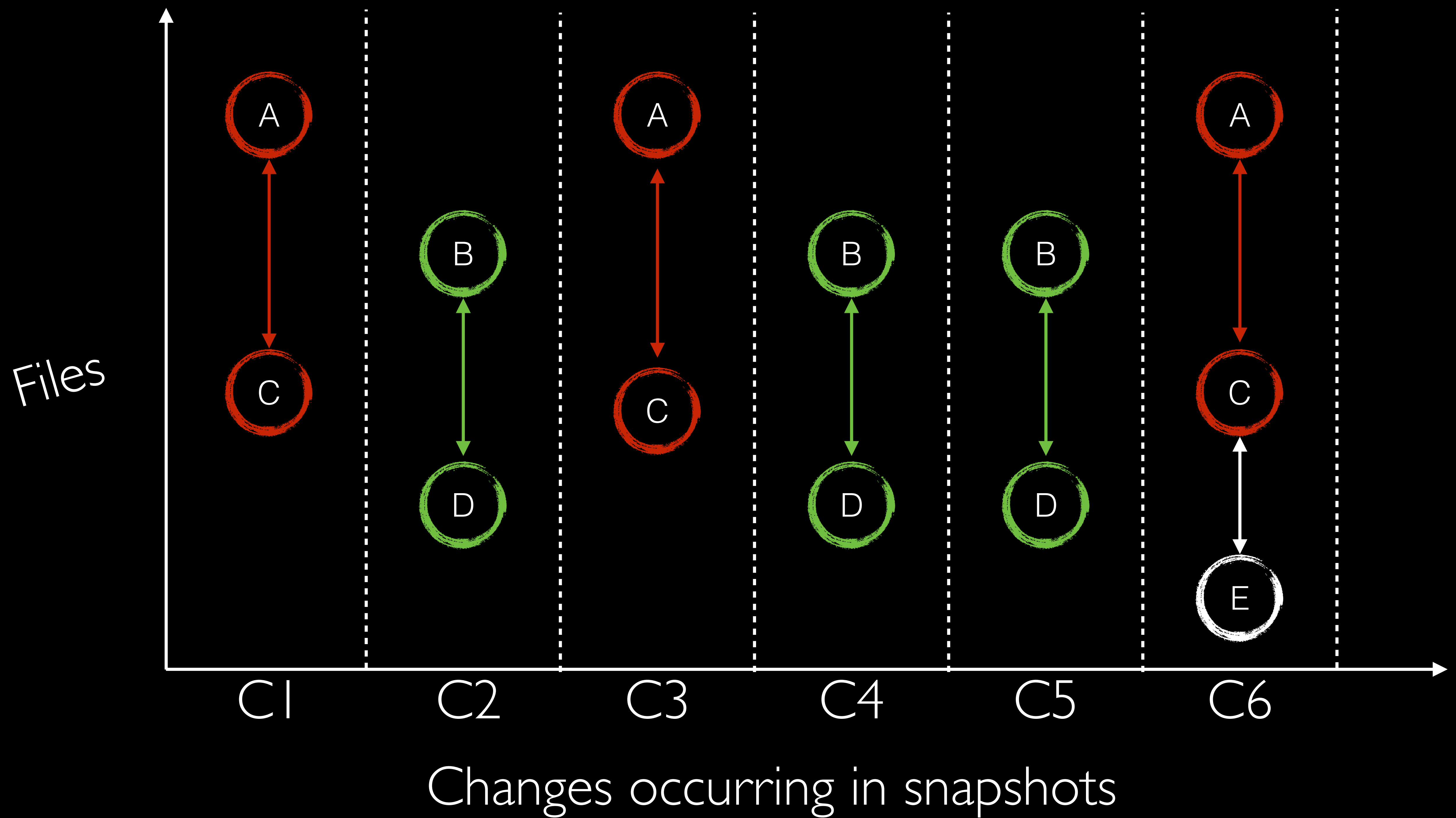


Analysis of change frequency of some specific entities

Association Rule Mining



Association Rule Mining



Code Smells Detector

divergent change

A class is changed in different ways for different reasons

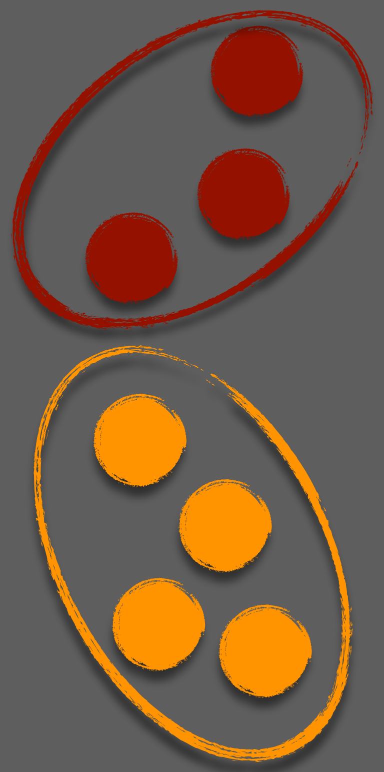
Solution:
Extract Class Refactoring

Detection

Classes containing at least two sets of methods such that:

(i) all methods in the set change together as detected by the association rules

(ii) each method in the set does not change with methods in other sets



Code Smells Detector

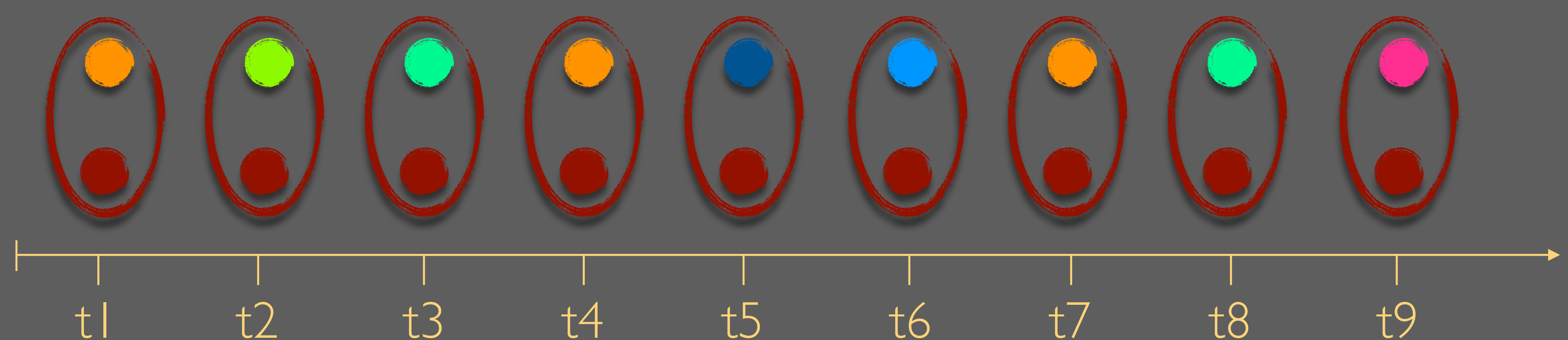
blob

A class implementing several responsibilities, having a large size, and dependencies with data classes

Solution: Extract Class refactoring

Detection

Blobs are identified as classes frequently modified in commits involving at least another class.



Evaluation

detection accuracy

20 open source systems

Comparing HIST with static code analysis technique on a manually built oracle

Evaluation

detection accuracy

20 open source systems

Comparing HIST with static code analysis technique on a manually built oracle

	HIST F-Measure	CA technique F-Measure
Shotgun Surgery	92%	0%
Parallel Inheritance	71%	9%
Divergent Change	82%	11%
Blob	64%	48%
Feature Envy	77%	68%

Evaluation

detection accuracy

20 open source systems

Comparing HIST with static code analysis technique on a manually built oracle

HIST and the CA techniques are highly complementary

Structural and Historical Analysis are only a part of the whole story

Toward a New Dimension of Code Smell Detection

The textual content of source code can provide useful hints for smell detection

```java
/* Insert a new user in the system.
 *  @param pUser: the user to insert.*/
public void insert(User pUser){

   connect = DBConnection.getConnection();

   String sql = "INSERT INTO USER"
      + "(login,first_name,last_name,password"
      + ",email,cell,id_parent) " + "VALUES ("
      + pUser.getLogin() + ","
      + pUser.getFirstName() + ","
      + pUser.getLastName() + ","
      + pUser.getPassword() + ","
      + pUser.getEMail() + ","
      + pUser.getCell() + ","
      + pUser.getIdParent() + ")";

   executeOperation(connect, sql);
 }
```

```java
/* Delete an user from the system.
 *  @param pUser: the user to delete.*/
public void delete(User pUser) {

   connect = DBConnection.getConnection();

   String sql = "DELETE FROM USER "
      + "WHERE id_user = "
      + pUser.getId();

   executeOperation(connect, sql);
 }
```

Indeed, source code vocabulary can be an useful additional source of information

```
/* Insert a new user in the system.
 *  @param pUser: the user to insert.*/
public void insert(User pUser){

   connect = DBConnection.getConnection();

   String sql = "INSERT INTO USER"
      + "(login,first_name,last_name,password"
      + ",email,cell,id_parent) " + "VALUES ("
      + pUser.getLogin() + ","
      + pUser.getFirstName() + ","
      + pUser.getLastName() + ","
      + pUser.getPassword() + ","
      + pUser.getEMail() + ","
      + pUser.getCell() + ","
      + pUser.getIdParent() + ")";

   executeOperation(connect, sql);
 }
```

```
/* Delete an user from the system.
 *  @param pUser: the user to delete.*/
public void delete(User pUser) {

   connect = DBConnection.getConnection();

   String sql = "DELETE FROM USER "
      + "WHERE id_user = "
      + pUser.getId();

   executeOperation(connect, sql);
 }
```

Extracting and Normalizing Text

```
private Connection connect = DBConnection.getConnection();
```

Extracting and Normalizing Text

```
private Connection connect = DBConnection.getConnection();
```

Separating Composed Identifiers

```
private Connection connect = DB Connection.get Connection();
```

Extracting and Normalizing Text

```
private Connection connect = DBConnection.getConnection();
```

Separating Composed Identifiers

```
private Connection connect = DB Connection.get Connection();
```

Lower Case Reduction

```
private connection connect = db connection.get connection();
```

Extracting and Normalizing Text

```
private Connection connect = DBConnection.getConnection();
```

Separating Composed Identifiers

```
private Connection connect = DB Connection.get Connection();
```

Lower Case Reduction

```
private connection connect = db connection.get connection();
```

Removing Special Characters, programming keywords, and common English terms

```
connection connect = db connection get connection
```

Extracting and Normalizing Text

```
private Connection connect = DBConnection.getConnection();
```

Separating Composed Identifiers

```
private Connection connect = DB Connection.get Connection();
```

Lower Case Reduction

```
private connection connect = db connection.get connection();
```

Removing Special Characters, programming keywords, and common English terms

```
connection connect = db connection get connection
```

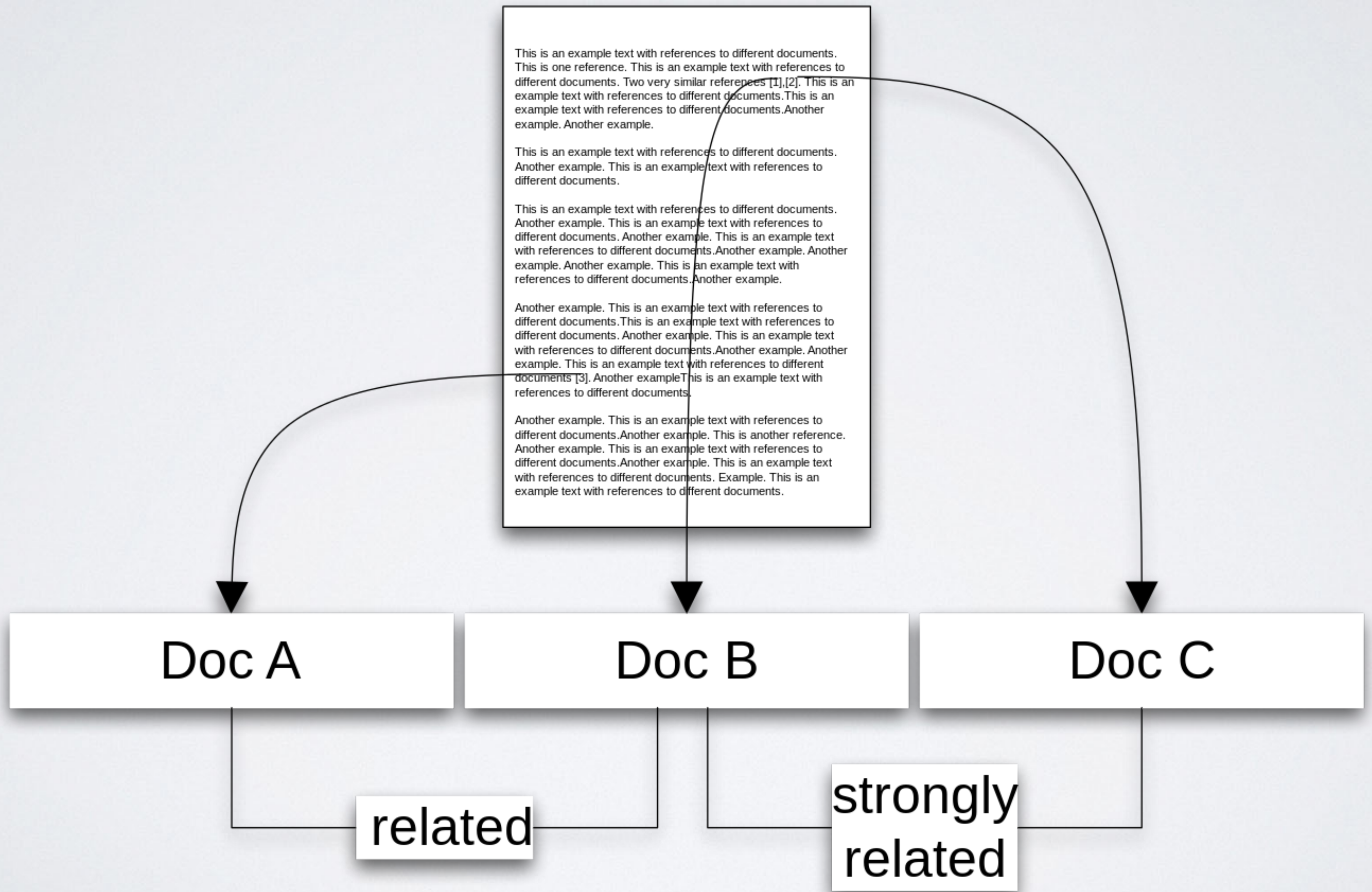
Stemming

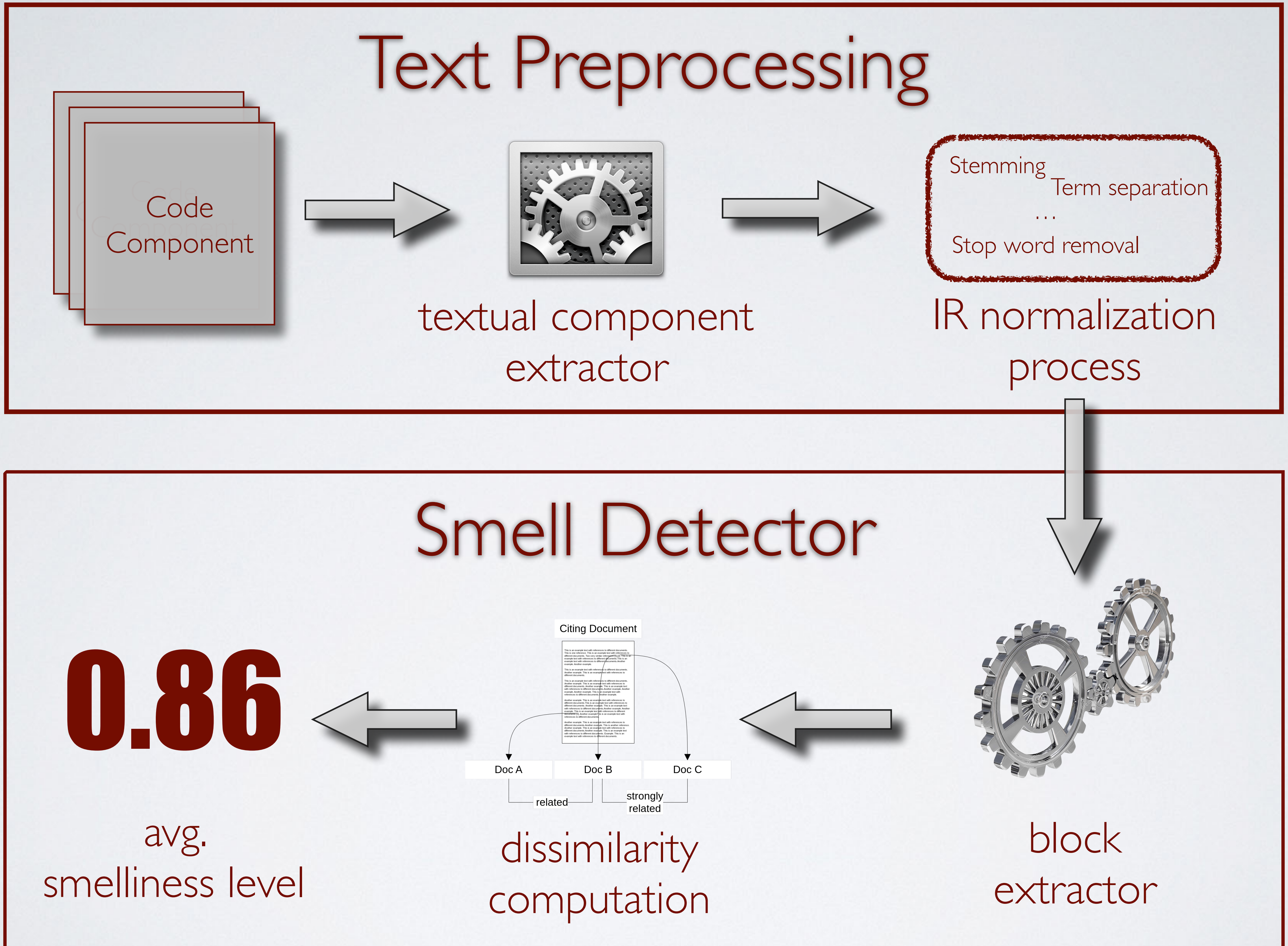
```
connect connect = db connect get connect
```

TACO

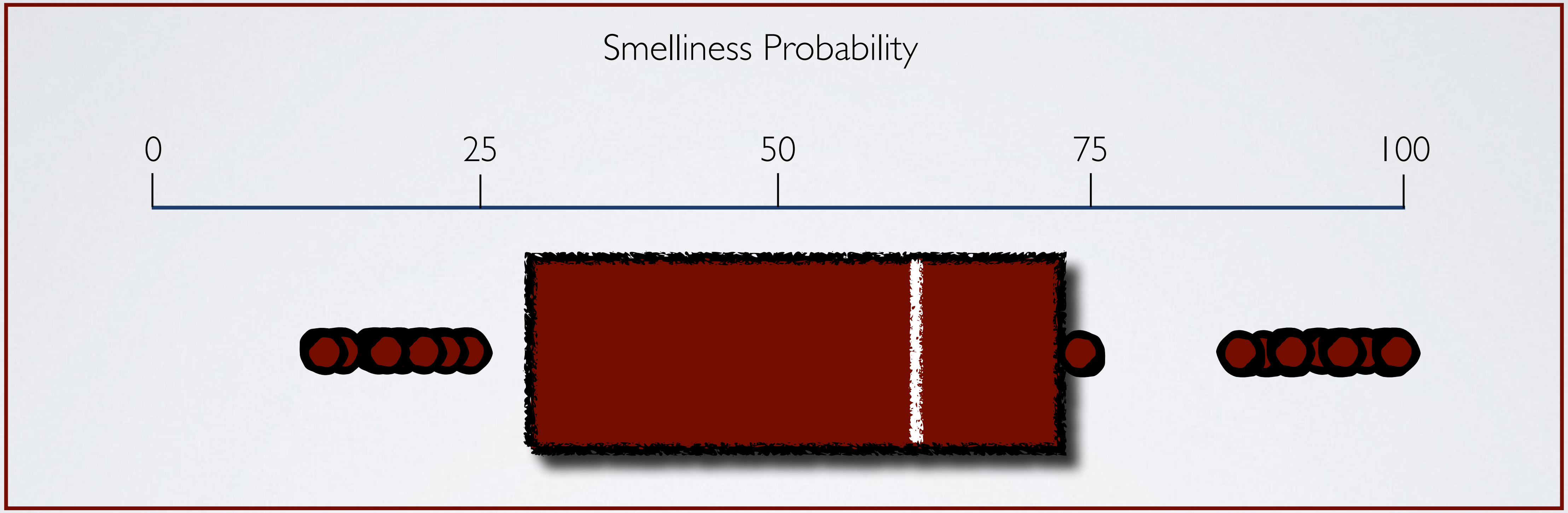
Textual Analysis for Code smell detectiOn

We believe that code affected by a smell contains unrelated textual content




Text Preprocessing
Code Component
textual component extractor
Stemming
Term separation
...
Stop word removal
IR normalization process
Smell Detector
block extractor
Citing Document
Doc A
Doc B
Doc C
related
strongly related
dissimilarity computation
0.86
avg. smelliness level

To detect smells, we need a threshold over the probability distribution



As cut point, we select the median of the non-null values of the smelliness

TACO can identify 5 different code smells characterized by promiscuous responsibilities

Long Method

Blob

Promiscuous Package

TACO can detect 5 different code smells characterized by promiscuous responsibilities

Long Method

Blob

Promiscuous Package

Feature Envy

Misplaced Class

Detecting Long Method instances

```
public void insert(User pUser){
  connect = DBConnection.getConnection();

  String sql = "INSERT INTO USER"
      + "(login,first_name,last_name,password"
      + ",email,cell,id_parent) " + "VALUES ("
      + pUser.getLogin() + ","
      + pUser.getFirstName() + ","
      + pUser.getLastName() + ","
      + pUser.getPassword() + ","
      + pUser.getEMail() + ","
      + pUser.getCell() + ","
      + pUser.getIdParent() + ")";

  String sql = "DELETE FROM USER "
      + "WHERE id_user = "
      + pUser.getId();
```

X. Whang, L. Pollock, K. Shanker
"*Automatic Segmentation of Method Code Into Meaningful Blocks: Design and Evaluation*"
JSEP 2013

Detecting Long Method instances

```
public void insert(User pUser){
  connect = DBConnection.getConnection();

  String sql = "INSERT INTO USER"
     + "(login,first_name,last_name,password"
     + ",email,cell,id_parent) " + "VALUES ("
     + pUser.getLogin() + ","
     + pUser.getFirstName() + ","
     + pUser.getLastName() + ","
     + pUser.getPassword() + ","
     + pUser.getEMail() + ","
     + pUser.getCell() + ","
     + pUser.getIdParent() + ")";

  String sql = "DELETE FROM USER "
     + "WHERE id_user = "
     + pUser.getId();
```

X. Whang, L. Pollock, K. Shanker
"*Automatic Segmentation of Method Code Into Meaningful Blocks: Design and Evaluation*"
JSEP 2013

Detecting Long Method instances

```
public void insert(User pUser){

  connect = DBConnection.getConnection();

  String sql = "INSERT INTO USER"
     + "(login,first_name,last_name,password"
     + ",email,cell,id_parent) " + "VALUES ("
     + pUser.getLogin() + ","
     + pUser.getFirstName() + ","
     + pUser.getLastName() + ","
     + pUser.getPassword() + ","
     + pUser.getEMail() + ","
     + pUser.getCell() + ","
     + pUser.getIdParent() + ")";

  String sql = "DELETE FROM USER "
     + "WHERE id_user = "
     + pUser.getId();
```

Method Cohesion Computation

X. Whang, L. Pollock, K. Shanker
"Automatic Segmentation of Method Code Into Meaningful Blocks: Design and Evaluation"
JSEP 2013

Detecting Long Method instances

```
public void insert(User pUser){
  connect = DBConnection.getConnection();

  String sql = "INSERT INTO USER"
      + "(login,first_name,last_name,password"
      + ",email,cell,id_parent) " + "VALUES ("
      + pUser.getLogin() + ","
      + pUser.getFirstName() + ","
      + pUser.getLastName() + ","
      + pUser.getPassword() + ","
      + pUser.getEMail() + ","
      + pUser.getCell() + ","
      + pUser.getIdParent() + ")";

  String sql = "DELETE FROM USER "
      + "WHERE id_user = "
      + pUser.getId();
```

Method Cohesion Computation

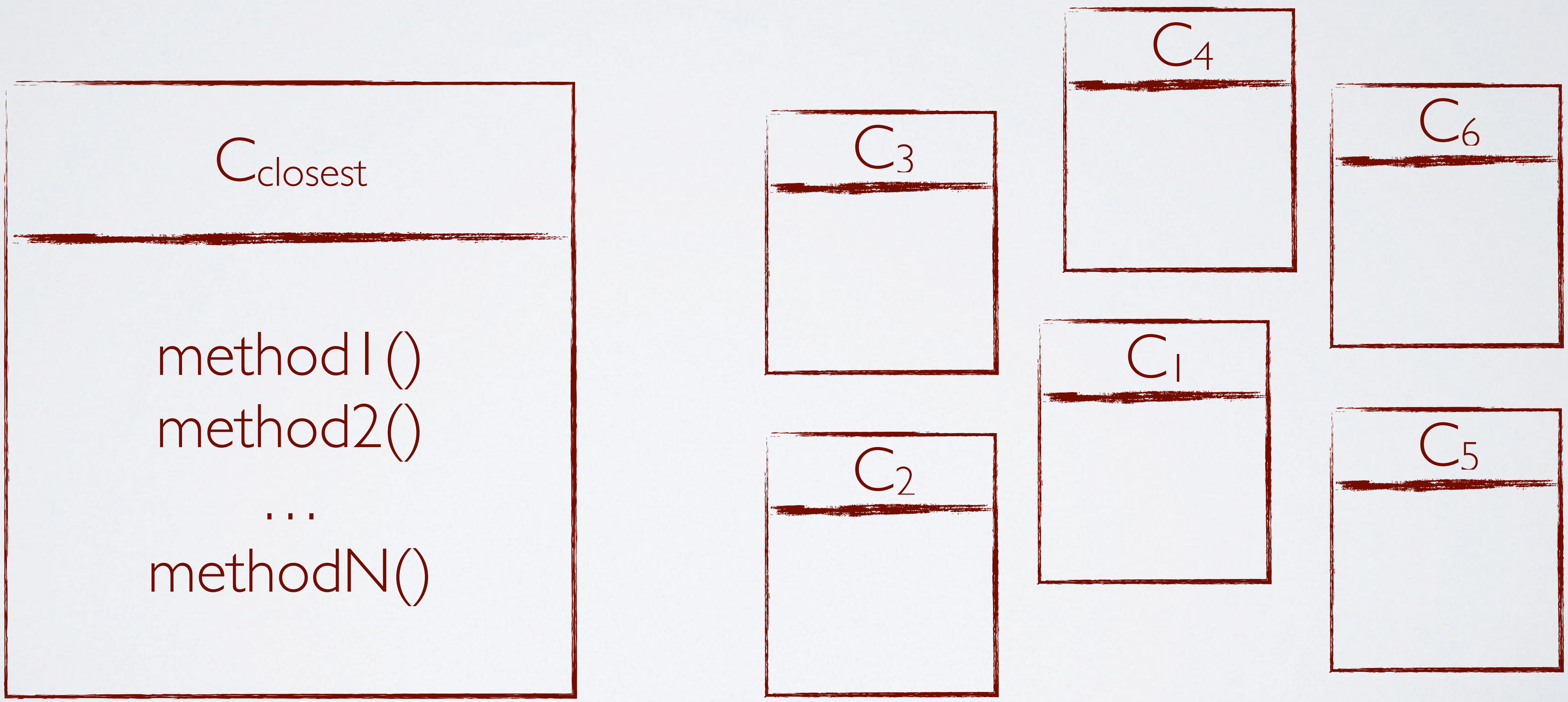
Long Method Probability Computation

X. Whang, L. Pollock, K. Shanker
"*Automatic Segmentation of Method Code Into Meaningful Blocks: Design and Evaluation*"
JSEP 2013

Detecting Feature Envy instances

Detecting Feature Envy instances

Extracting the class $C_{closest}$ having the highest textual similarity with M_i



Detecting Feature Envy instances

Feature Envy Probability Computation

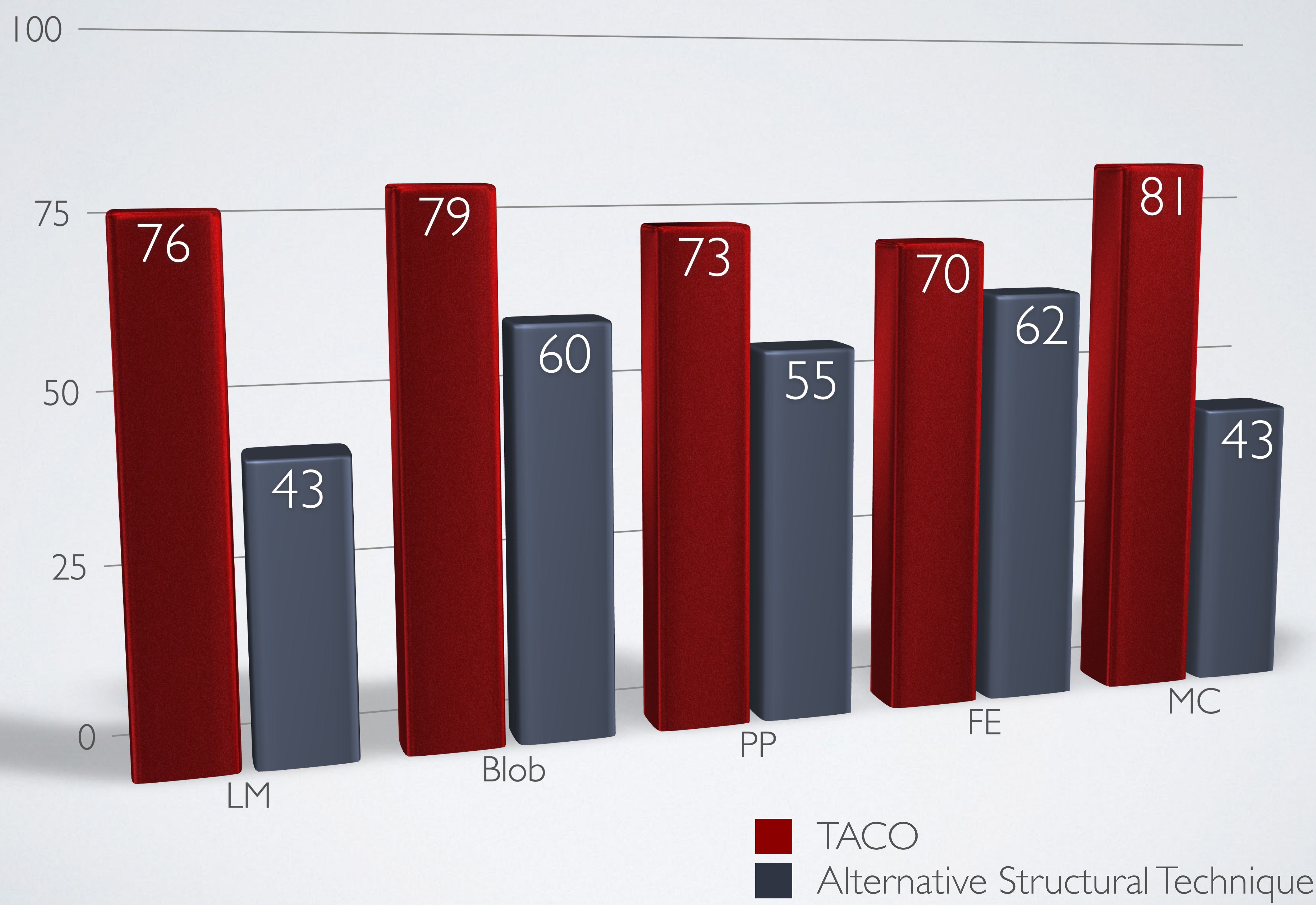
$C_{closest}$

method1()
method2()
...
methodN()

C_O

method1()
method2()
...
methodN()

TACO - Evaluating its performance



TACO - Evaluating its performance

+22%

on average in terms of F-Measure

TACO - Evaluating its performance

Method: findTypesAndPackages()

Class: CompletionEngine - Eclipse Core

Goal: Discover the classes and the packages of a given project

TACO - Evaluating its performance

Method: findTypesAndPackages()

Class: CompletionEngine - Eclipse Core

Goal: Discover the classes and the packages of a given project

65

lines of code

TACO - Evaluating its performance

Method: findTypesAndPackages()

Class: CompletionEngine - Eclipse Core

Goal: Discover the classes and the packages of a given project

A Structural Approach cannot detect the smell!

65

lines of code

TACO - Evaluating its performance

Method: findTypesAndPackages()

Class: CompletionEngine - Eclipse Core

Goal: Discover the classes and the packages of a given project

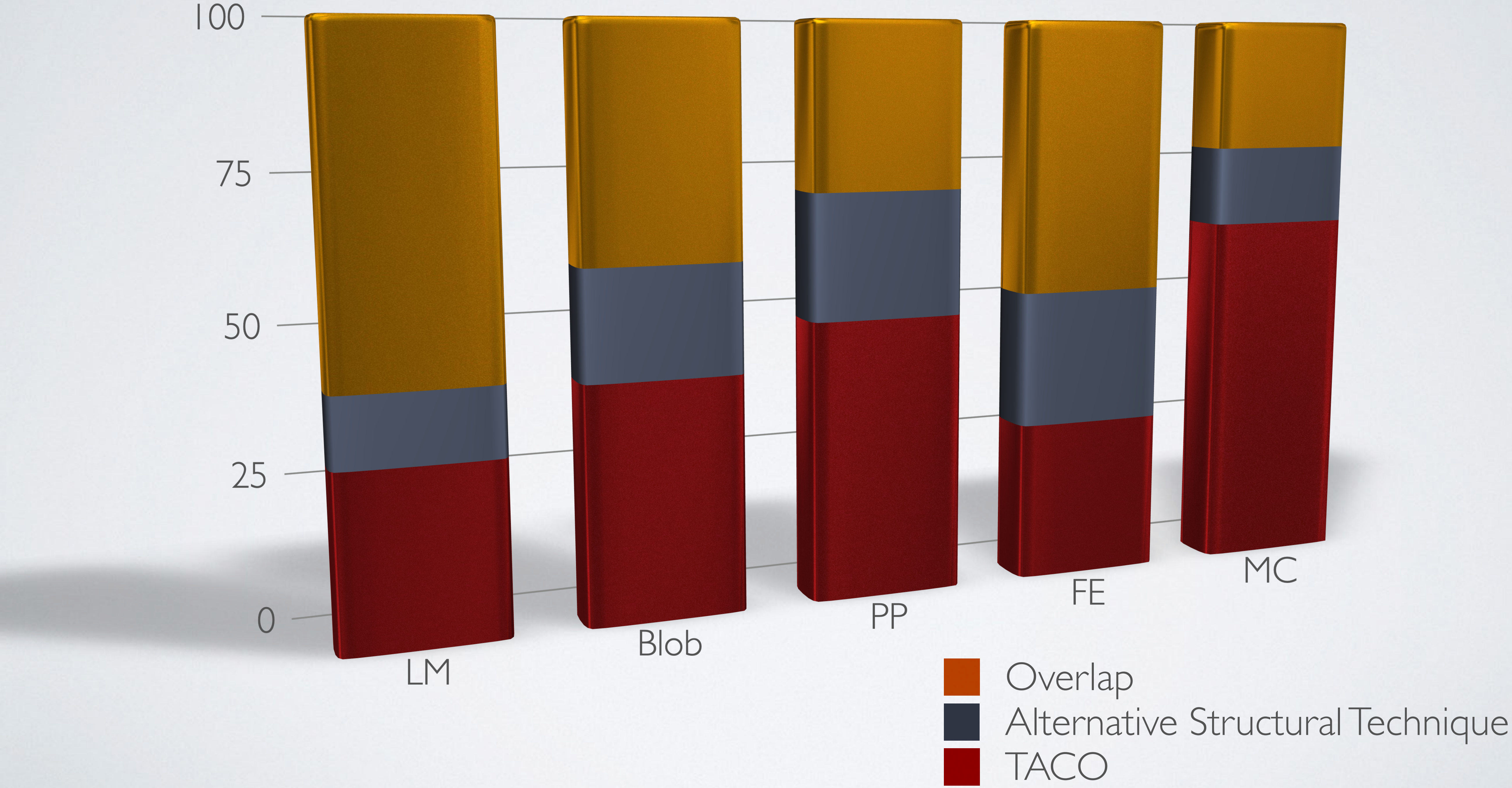
65

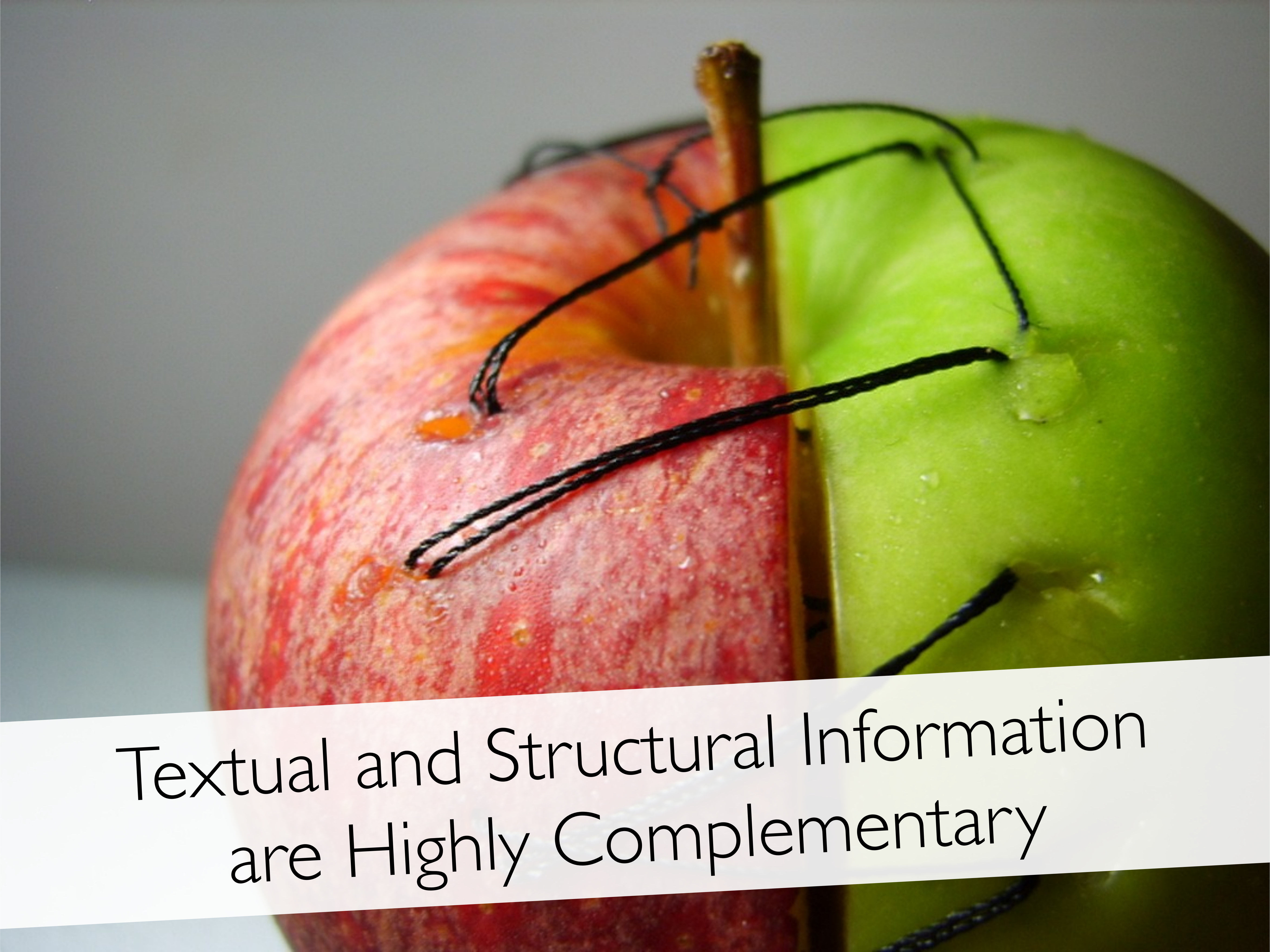
lines of code

A Structural Approach cannot detect the smell!

TACO, instead, is able to detect a Long Method instance

TACO - Evaluating complementarity with structural approaches




Textual and Structural Information
are Highly Complementary

Toward a combination of code smell detection techniques?

Code and **Test Smells**

Understanding and Detecting Them

Smells in Test Code

Refactoring Test Code

Arie van Deursen **Leon Moonen**
CWI
The Netherlands
http://www.cwi.nl/~{arie,leon}/
{arie,leon}@cwi.nl

Alex van den Bergh **Gerard Kok**
Software Improvement Group
The Netherlands
http://www.software-improvers.com/
{alex,gerard}@software-improvers.com

ABSTRACT

Two key aspects of extreme programming (XP) are unit testing and merciless refactoring. Given the fact that the ideal test code / production code ratio approaches 1:1, it is not surprising that unit tests are being refactored. We found that refactoring test code is different from refactoring production code in two ways: (1) there is a distinct set of bad smells involved, and (2) improving test code involves additional test-specific refactorings. To share our experiences with other XP practitioners, we describe a set of bad smells that indicate trouble in test code, and a collection of test refactorings to remove these smells.

Keywords
Refactoring, unit testing, extreme programming.

1 INTRODUCTION

"If there is a technique at the heart of *extreme programming* (XP), it is unit testing" [1]. As part of their programming activity, XP developers write and maintain (white box) unit tests continually. These tests are automated, written in the same programming language as the production code, considered an explicit part of the code, and put under revision control.

The XP process encourages writing a test class for every class in the system. Methods in these test classes are used to verify complicated functionality and unusual circumstances. Moreover, they are used to document code by explicitly indicating what the expected results of a method should be for typical cases. Last but not least, tests are added upon receiving a bug report to check for the bug and to check the bug fix [2]. A typical test for a particular method includes: (1) code to set up the fixture (the data used for testing), (2) the call of the method, (3) a comparison of the actual results with the expected values, and (4) code to tear down the fixture. Writing tests is usually supported by frameworks such as *JUnit* [3].

The test code / production code ratio may vary from project to project, but is ideally considered to approach a ratio of 1:1. In our project we currently have a 2:3 ratio, although others have reported a lower ratio[1]. One of the corner stones of XP is that having many tests available helps the developers to overcome their fear for change: the tests will provide immediate feedback if the system gets broken at a critical place. The downside of having many tests, however, is that changes in functionality will typically involve changes in the test code as well. The more test code we get, the more important it becomes that this test code is as easily modifiable as the production code.

The key XP practice to keep code flexible is "refactor mercilessly": transforming the code in order to bring it in the simplest possible state. To support this, a catalog of "code smells" and a wide range of refactorings is available, varying from simple modifications up to ways to introduce design patterns systematically in existing code [5].

When trying to apply refactorings to the test code of our project we discovered that refactoring test code is different from refactoring production code. Test code has a distinct set of smells, dealing with the ways in which test cases are organized, how they are implemented, and how they interact with each other. Moreover, improving test code involves a mixture of refactorings from [5] specialized to test code improvements, as well as a set of additional refactorings, involving the modification of test classes, ways of grouping test cases, and so on.

The goal of this paper is to share our experience in improving our test code with other XP practitioners. To that end, we describe a set of *test smells* indicating trouble in test code, and a collection of *test refactorings* explaining how to overcome some of these problems through a simple program modification.

This paper assumes some familiarity with the xUnit framework [3] and refactorings as described by Fowler [5]. We will refer to refactorings described in this book using *Name*

[1] This project started a year ago and involves the development of a product called DocGen [4]. Development is done by a small team of five people using XP techniques. Code is written in Java and we use the JUnit framework for unit testing.

"Test Smells represent a set of a poor design solutions to write tests"

[Van Deursen et al. - XP 2001]

11

test smells related to the way developers write test fixtures and test cases

Smells in Test Code

```
public void test12 () throws Throwable {
JSTerm jSTerm0 = new JSTerm();
jSTerm0.makeVariable () ;
jSTerm0.add((Object) ””);
jSTerm0.matches(jSTerm0);
assertEquals (false, jSTerm0.isGround ());
assertEquals(true, jSTerm0.isVariable());
}
```

Smells in Test Code

```
public void test12 () throws Throwable {
JSTerm jSTerm0 = new JSTerm();
jSTerm0.makeVariable () ;
jSTerm0.add((Object) ””);
jSTerm0.matches(jSTerm0);
assertEquals (false, jSTerm0.isGround ());
assertEquals(true, jSTerm0.isVariable());
}
```

The test method checks the production method isGround()

Smells in Test Code

```
public void test12 () throws Throwable {
JSTerm jSTerm0 = new JSTerm();
jSTerm0.makeVariable () ;
jSTerm0.add((Object) ””);
jSTerm0.matches(jSTerm0);
assertEquals (false, jSTerm0.isGround ());
assertEquals(true, jSTerm0.isVariable());
}
```

But also the production method isVariable()

Smells in Test Code

```
public void test12 () throws Throwable {
JSTerm jSTerm0 = new JSTerm();
jSTerm0.makeVariable () ;
jSTerm0.add((Object) ””);
jSTerm0.matches(jSTerm0);
assertEquals (false, jSTerm0.isGround ());
assertEquals(true, jSTerm0.isVariable());
}
```

This is an Eager Test, namely a test which checks more than one method of the class to be tested, making difficult the comprehension of the actual test target.

Smells in Test Code

A test case is affected by a Resource Optimism when it makes assumptions about the state or the existence of external resources, providing a non-deterministic result that depend on the state of the resources.

An Assertion Roulette comes from having a number of assertions in a test method that have no explanation. If an assertion fails, the identification of the assert that failed can be difficult.

Smells in Test Code

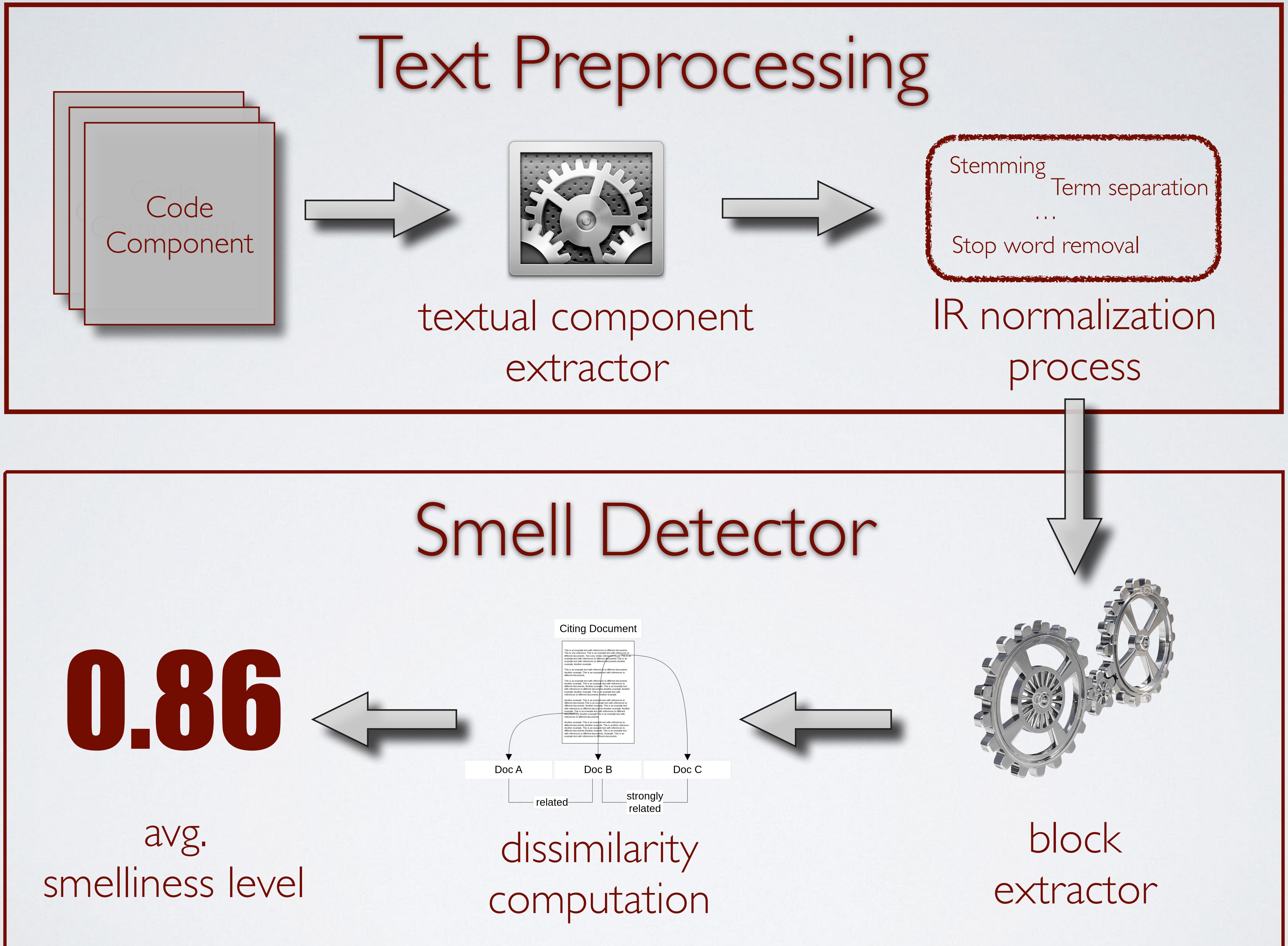
Tests affected by test smells are more change- and fault-prone than tests not participating in design flaws and affect the reliability of production code

In 54% of the cases, test code flakiness can be induced by the presence of some design flaw in test code

Detecting test smells using heuristics

```
public void test12 () throws Throwable {
JSTerm jSTerm0 = new JSTerm();
jSTerm0.makeVariable () ;
jSTerm0.add((Object) ””);
jSTerm0.matches(jSTerm0);
assertEquals (false, jSTerm0.isGround ());
assertEquals(true, jSTerm0.isVariable());
}
```

Test smell detected if the number of method calls > 3



Text Preprocessing
Code Component
textual component extractor
Stemming
Term separation
...
Stop word removal
IR normalization process
Smell Detector
block extractor
Citing Document
Doc A
Doc B
Doc C
related
strongly related
dissimilarity computation
0.86
avg. smelliness level

TASTE: Detecting test smells using the textual component of test code

test method m

…

A.x()

…

A.y()

TASTE: Detecting test smells using the textual component of test code

test method m

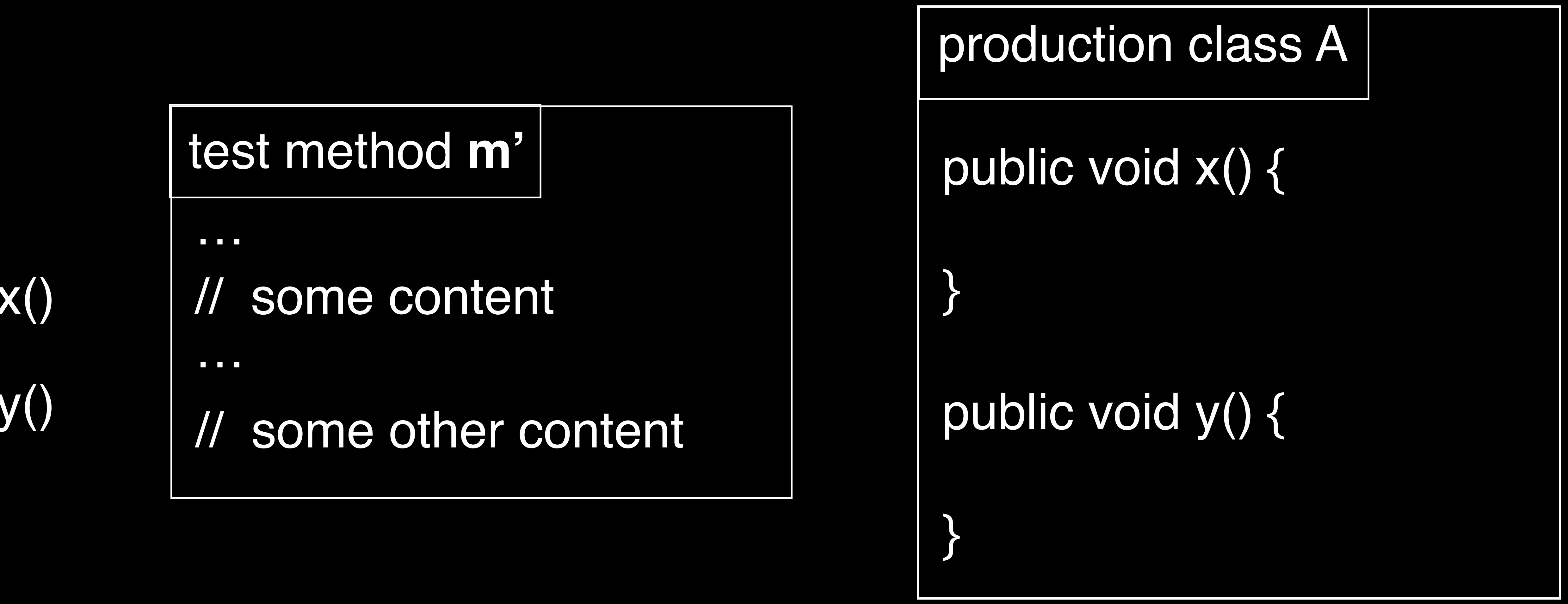
```
…
A.x()
…
A.y()
```

production class A

```
public void x() {
   //  some content
}

public void y() {
   //  some other content
}
```

TASTE: Detecting test smells using the textual component of test code



TASTE: Detecting test smells using the textual component of test code

x()

y()

test method m'

…

// some content

…

// some other content

IR normalization

TASTE: Detecting test smells using the textual component of test code

x()

y()

test method m'

…

// some content

…

// some other content

IR normalization

$$\underset{i \,!=\, j}{\text{mean}} \ \text{sim}(m'_i, m'_j)$$

TASTE: Detecting test smells using the textual component of test code

x()

y()

test method m’

…

// some content

…

// some other content

IR normalization

$$p_{ET}(t) = 1 - \underset{i\,!=\,j}{\mathrm{mean}}\ \mathrm{sim}(m'_i, m'_j)$$

TASTE: Detecting test smells using the textual component of test code

x()

y()

test method **m’**

…

// some content

…

// some other content

IR normalization

$p_{ET}(t) > 0.5$

Code and Test Smells
Understanding and Detecting Them

Fabio Palomba
Assistant Professor
University of Salerno (Italy)
https://fpalomba.github.io