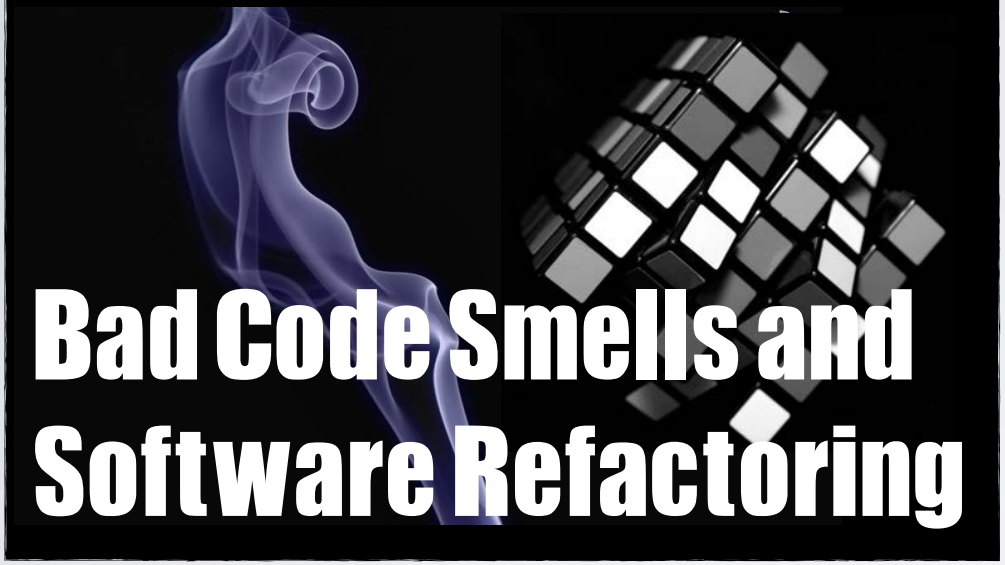
# Code Smell Identification and Refactoring Automation: Challenges, Solutions, and Open Issues

**Andrea De Lucia**
**University of Salerno**
**adelucia@unisa.it**

# Part I

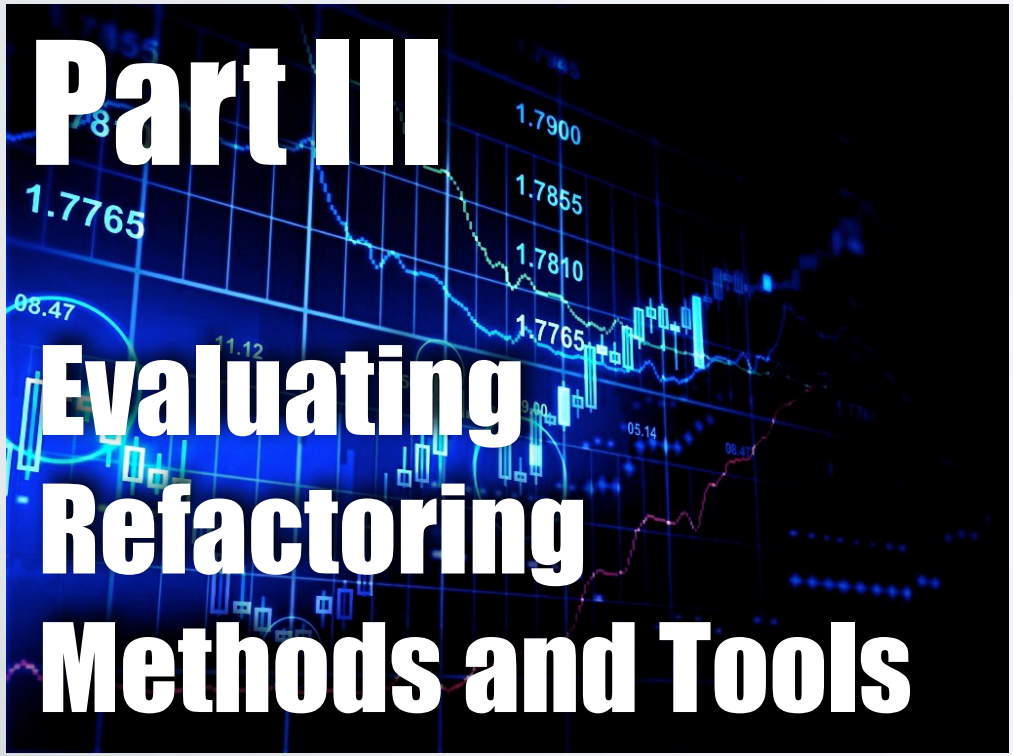


## Bad Code Smells and Software Refactoring

# Part II

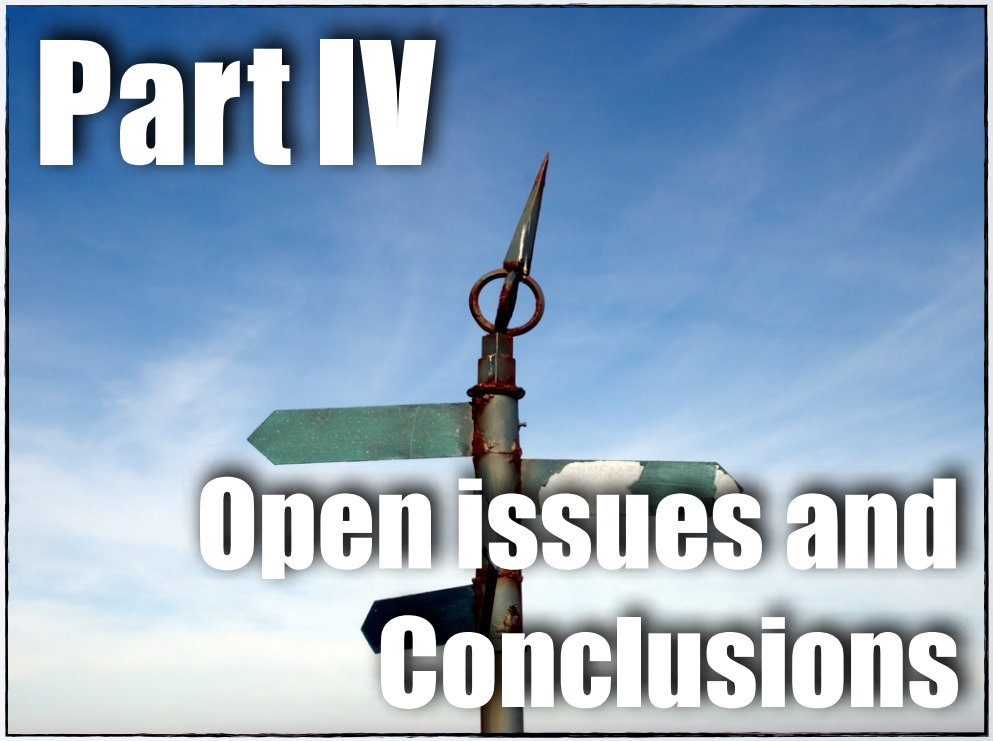
## The refactoring process

# Part III




## Evaluating Refactoring Methods and Tools

# Part IV

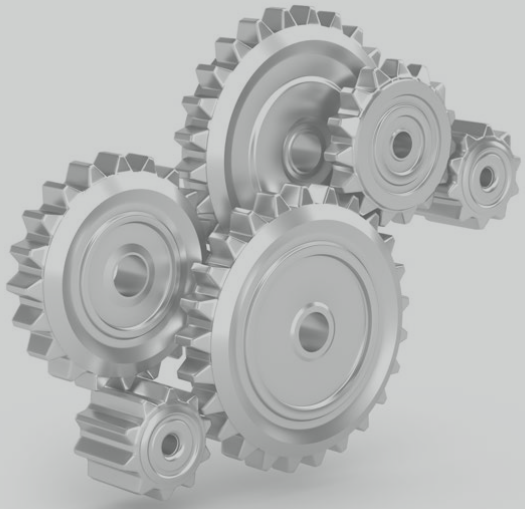


## Open issues and Conclusions

# Part I

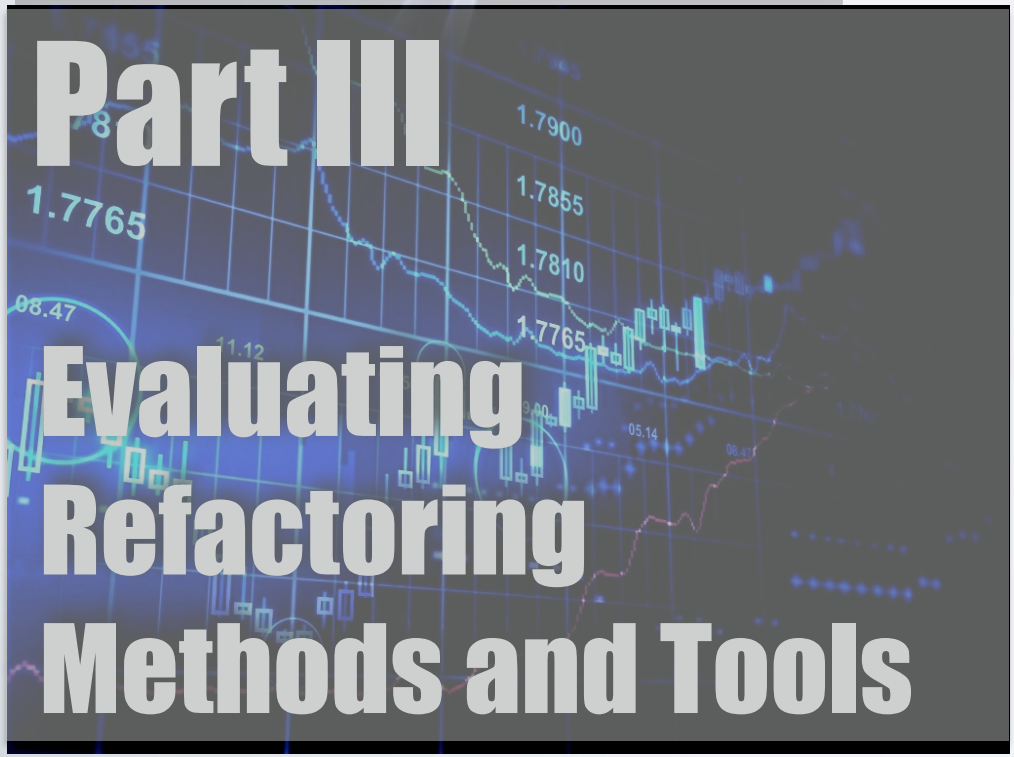
## Bad Code Smells and Software Refactoring

# Part II



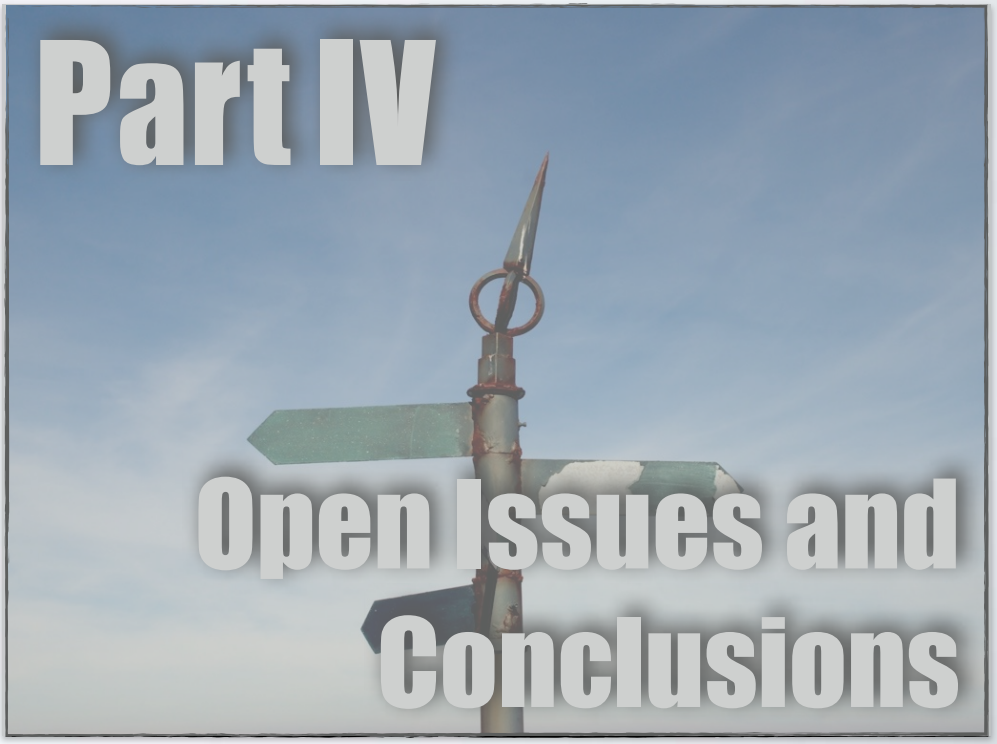
## The refactoring process

# Part III



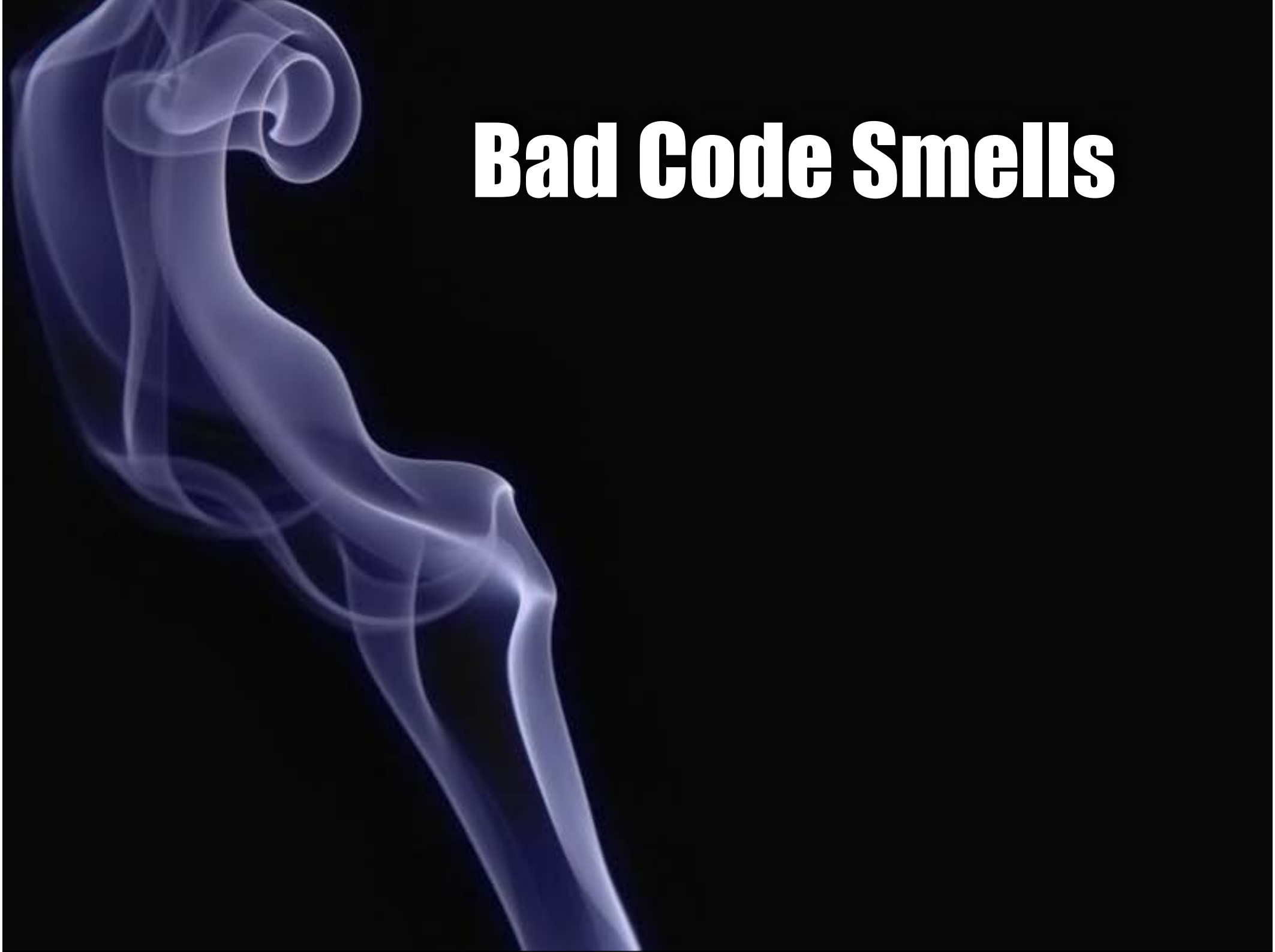

## Evaluating Refactoring Methods and Tools

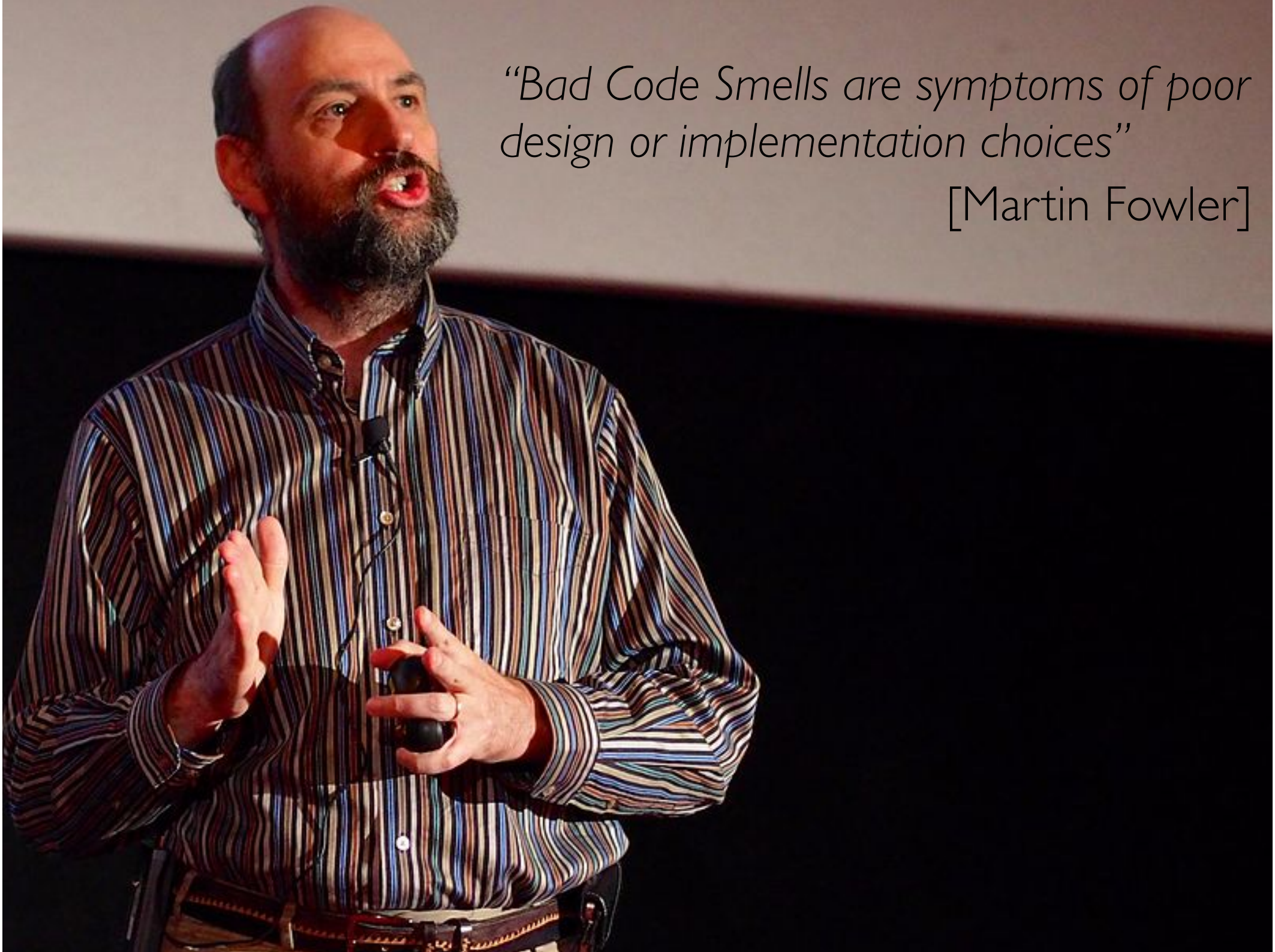
# Part IV



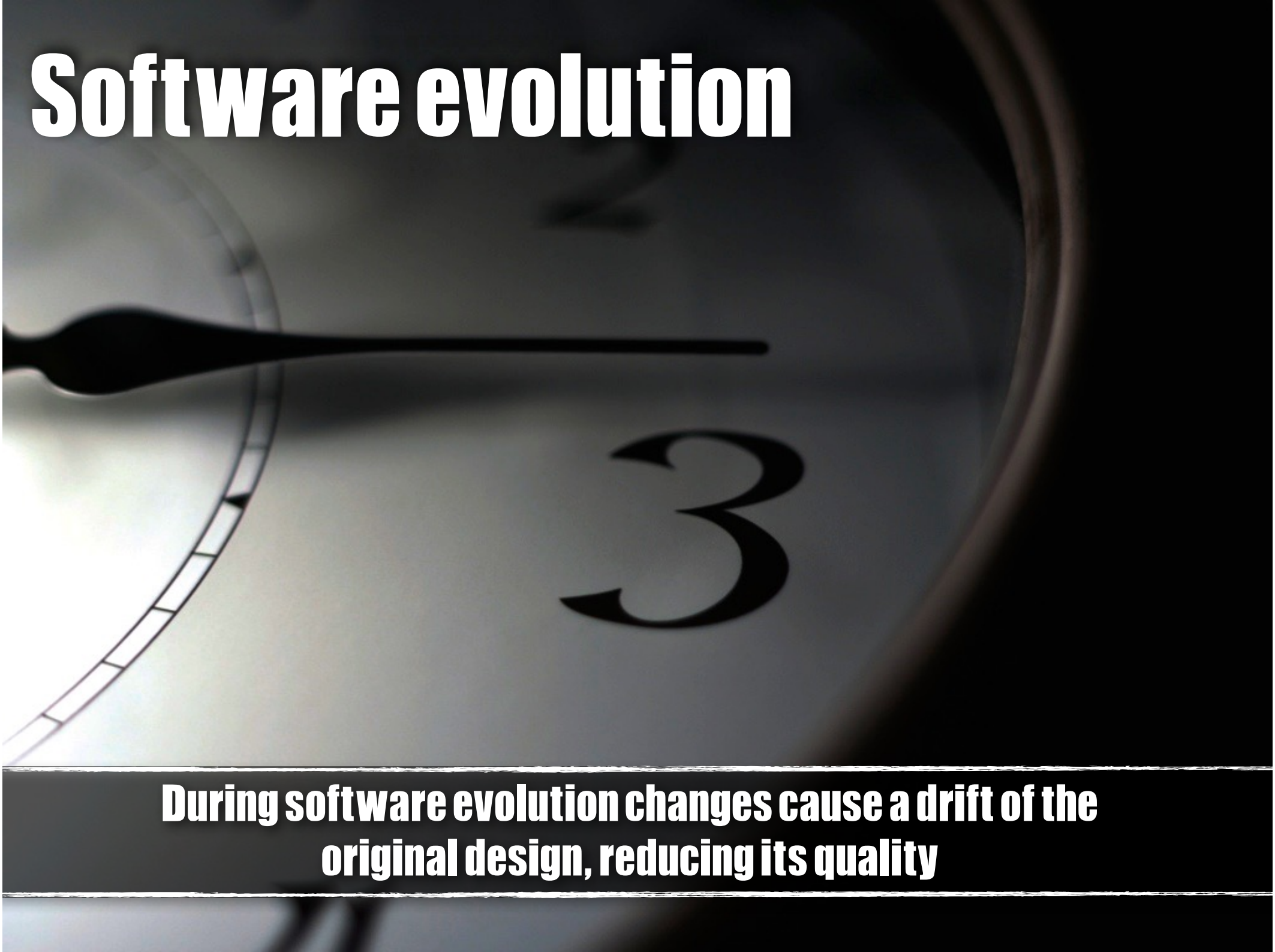
## Open Issues and Conclusions

# Bad Code Smells and Software Refactoring


Bad Code Smells


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

# Software evolution



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

# Low design quality ...

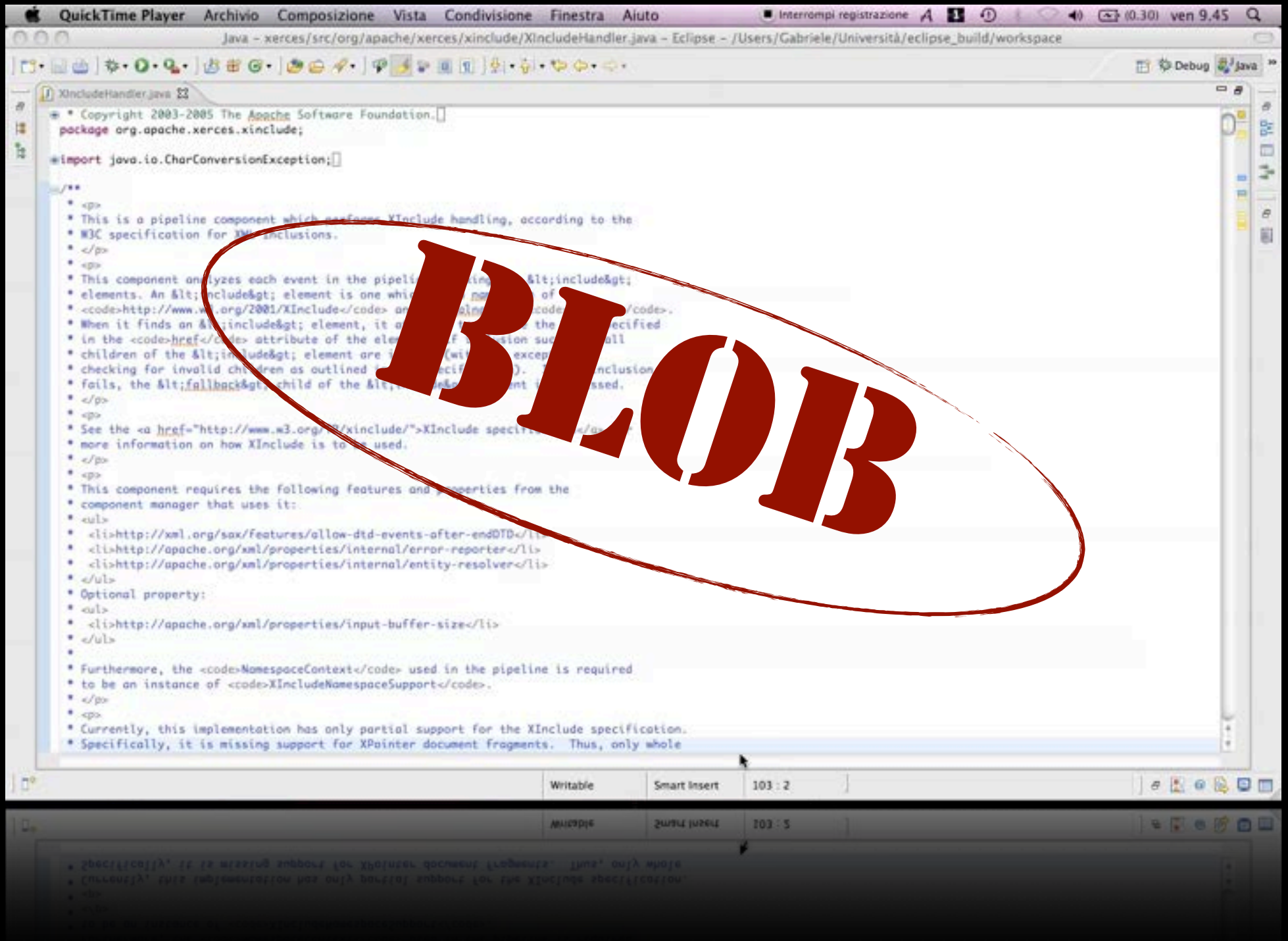
... has been associated with lower productivity, greater rework, and more significant efforts for developers


Victor R. Basili, Lionel Briand, and Walce´lio 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.


QuickTime Player Archivio Composizione Vista Condivisione Finestra Aiuto
Java – xerces/src/org/apache/xerces/xinclude/XIncludeHandler.java – Eclipse – /Users/Gabriele/Università/eclipse_build/workspace
XIncludeHandler.java
* Copyright 2003-2005 The Apache Software Foundation.
package org.apache.xerces.xinclude;
import java.io.CharConversionException;
BLOB
Writable
Smart Insert
103 : 2

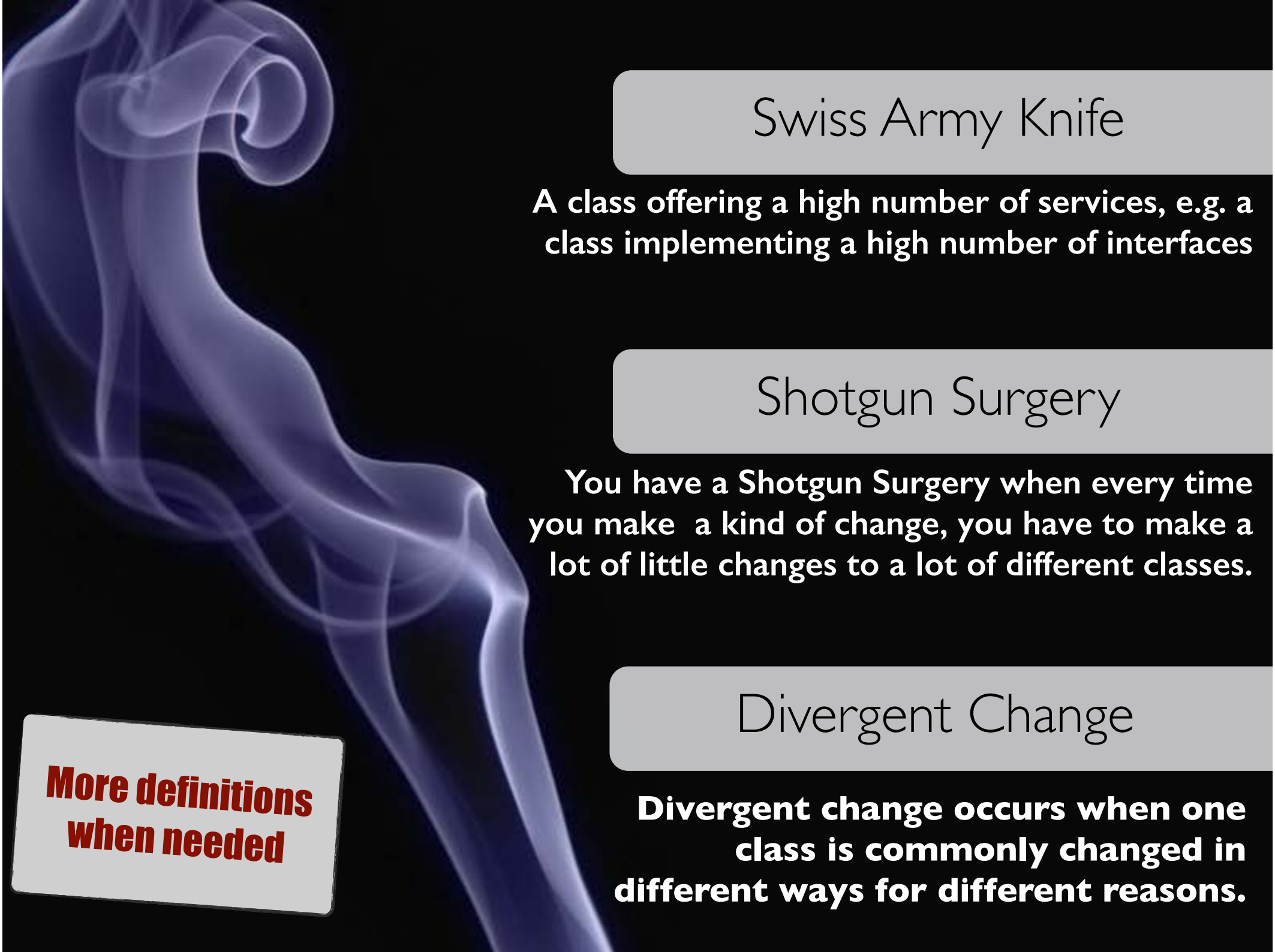
# Blob (or God Class)

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

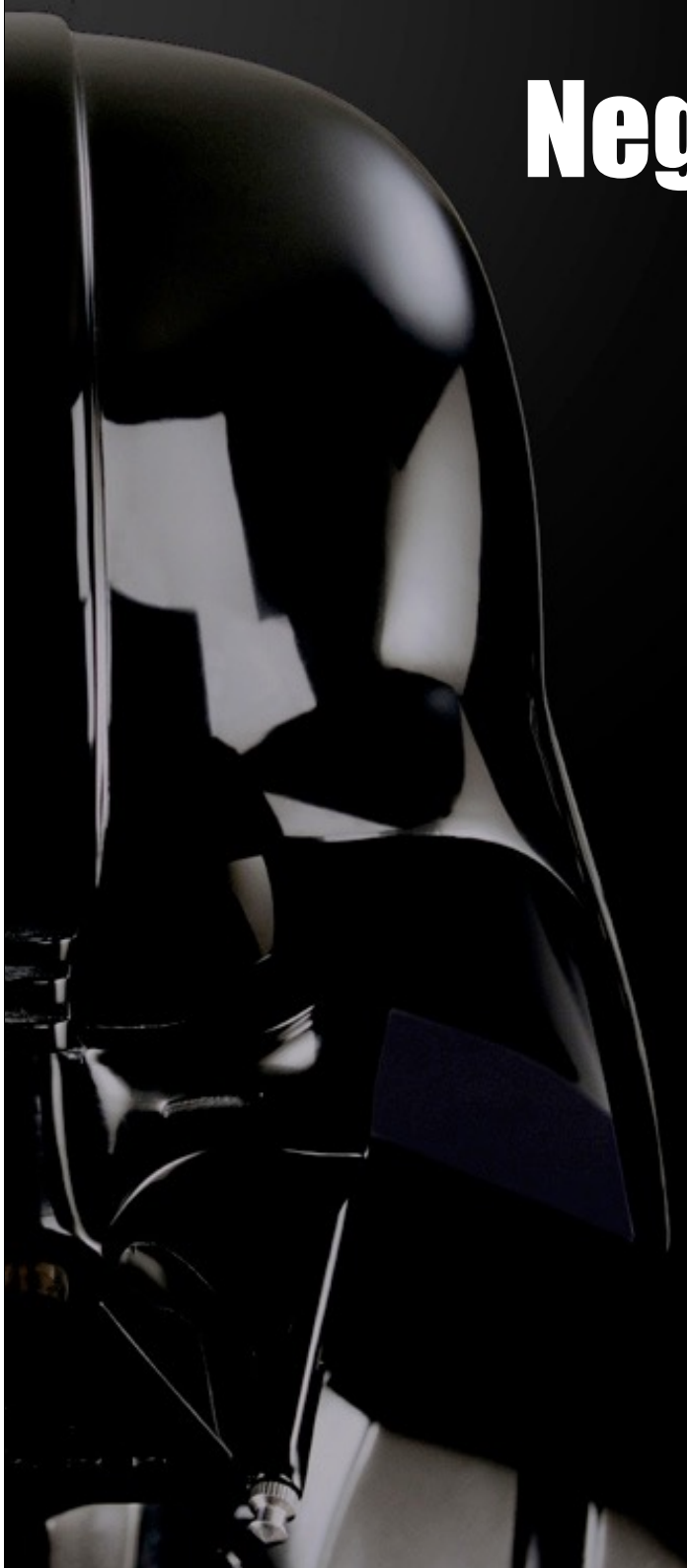
**[Martin Fowler]**

## Consequences

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


Swiss Army Knife
A class offering a high number of services, e.g. a class implementing a high number of interfaces
Shotgun Surgery
You have a Shotgun Surgery when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.
Divergent Change
Divergent change occurs when one class is commonly changed in different ways for different reasons.
More definitions when needed

# 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[1,2], Foutse Khomh[2], Yann-Gaël Guéhéneuc[3], Giuliano Antoniol[2]
[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 combination: 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 centralizes 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 summarized 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.

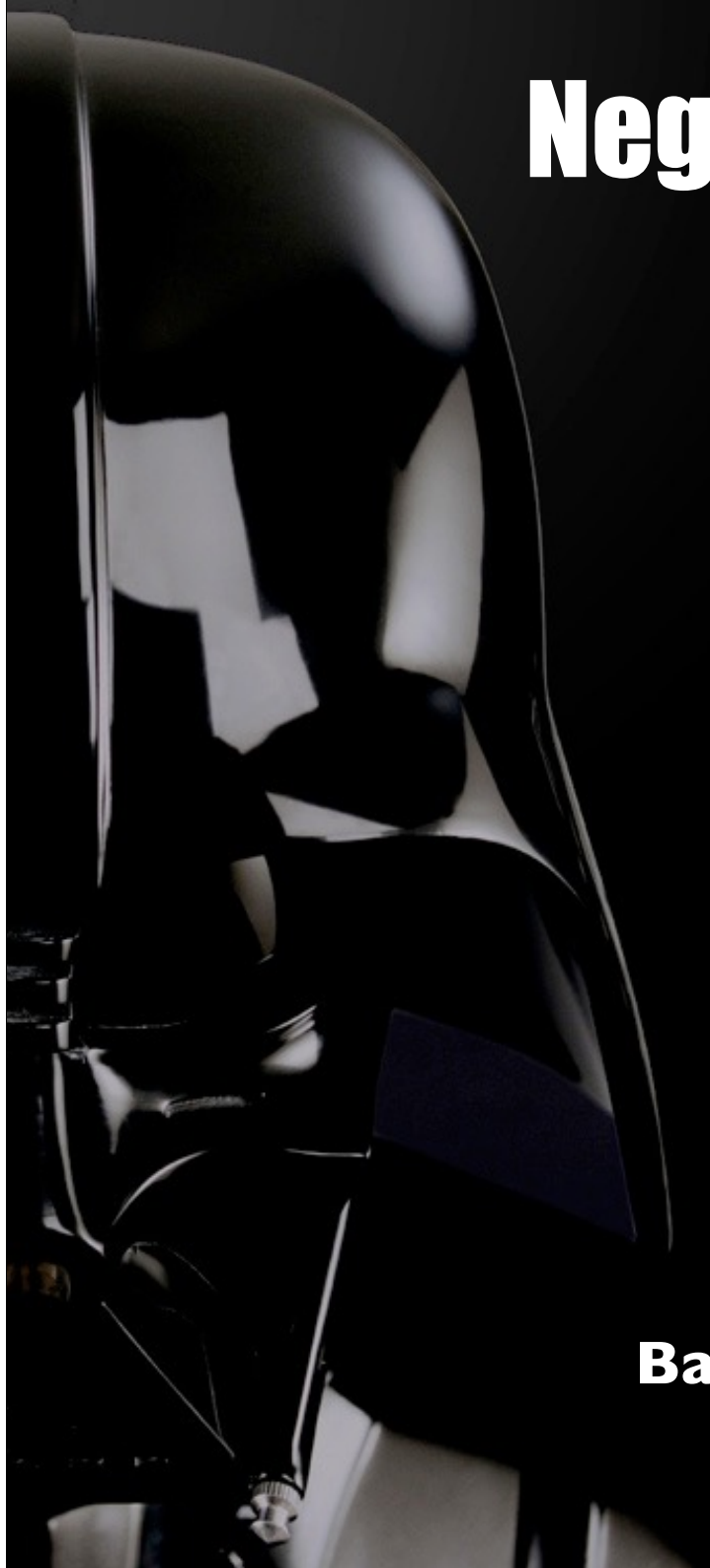
*Study*: We perform three experiments: we study whether systems with the antipatterns Blob, first, and the Spaghetti Code, second, are more difficult to understand than systems without any antipatterns. Third, we study whether systems with both Blob and Spaghetti Code are more difficult to understand 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



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 undergo 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

Springer

**Bad Smells increase change- and fault-pronene**

**[Khomh et al. EMSE 2012]**

# Negative Impact of Bad Smells

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

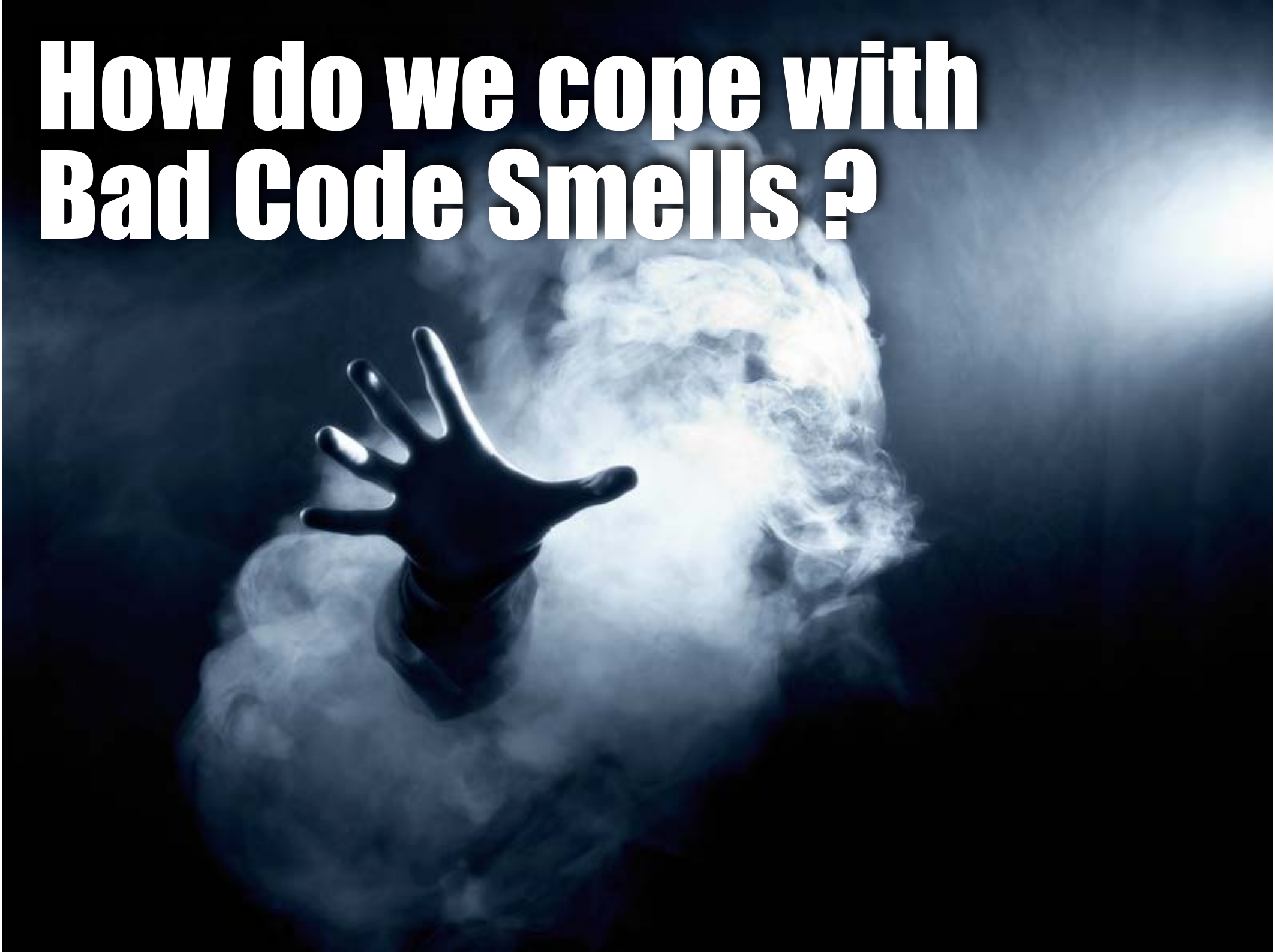
**SOFTWARE COMPLEXITY AND MAINTENANCE COSTS**

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

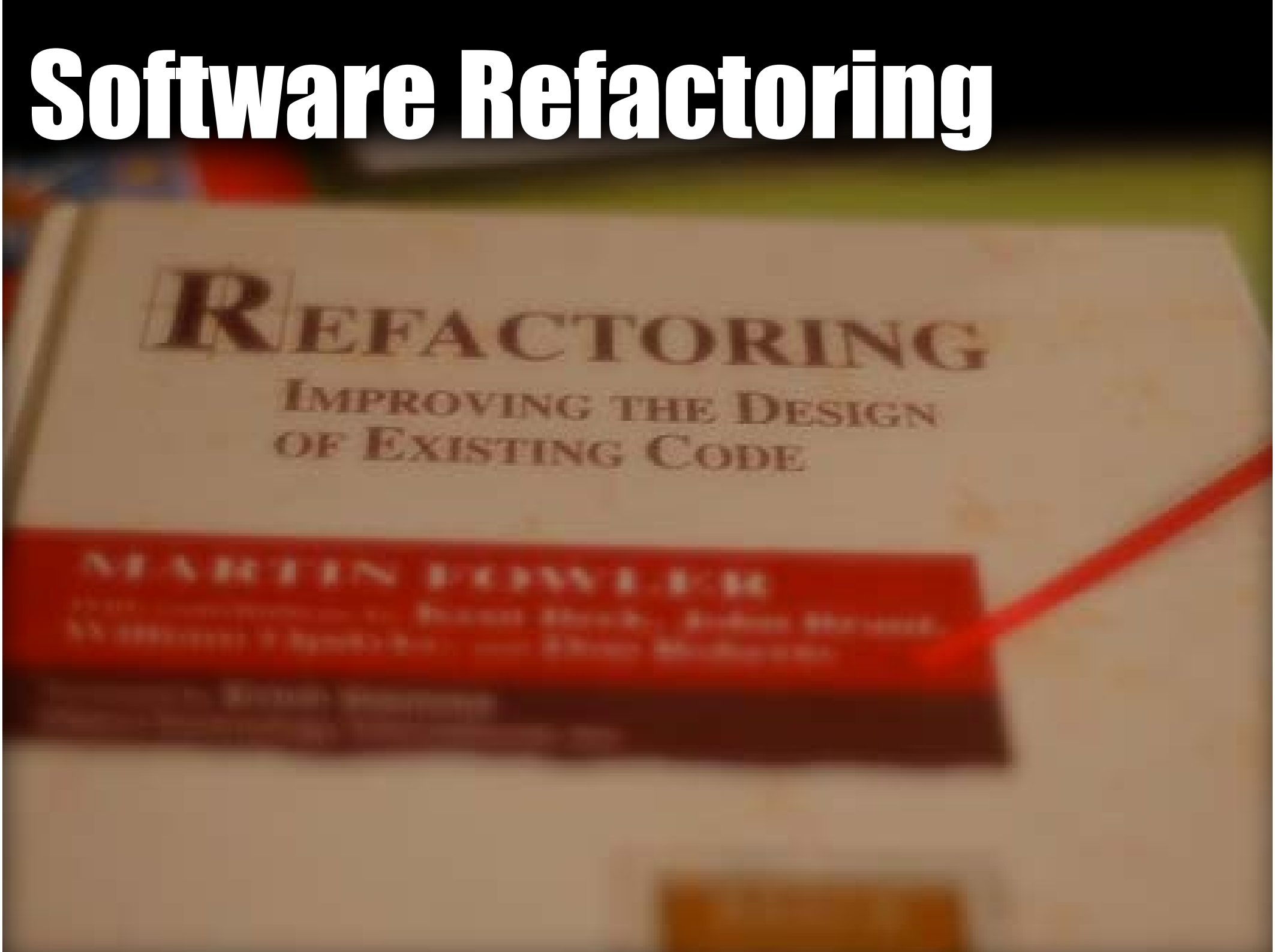
**Bad Smells increase maintenance costs**

**[Banker et al. Communications of the ACM]**

# How do we cope with Bad Code Smells ?



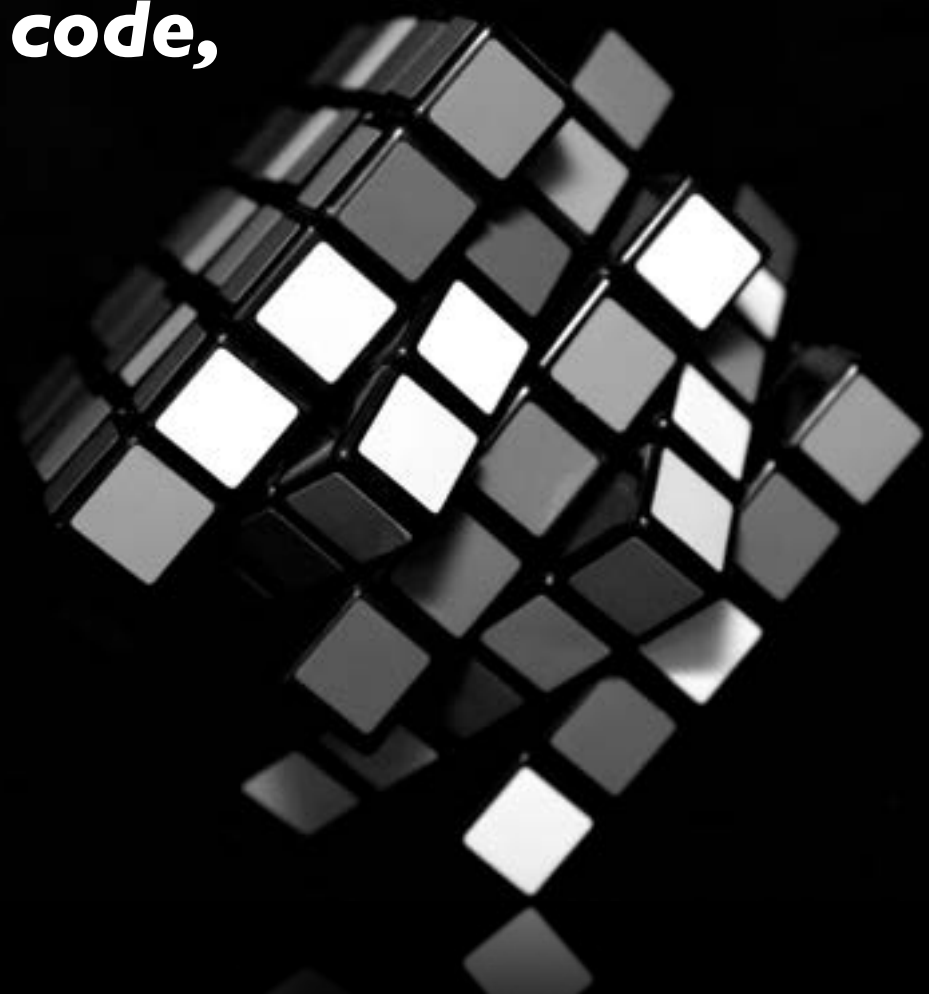
# Software Refactoring



# Software Refactoring

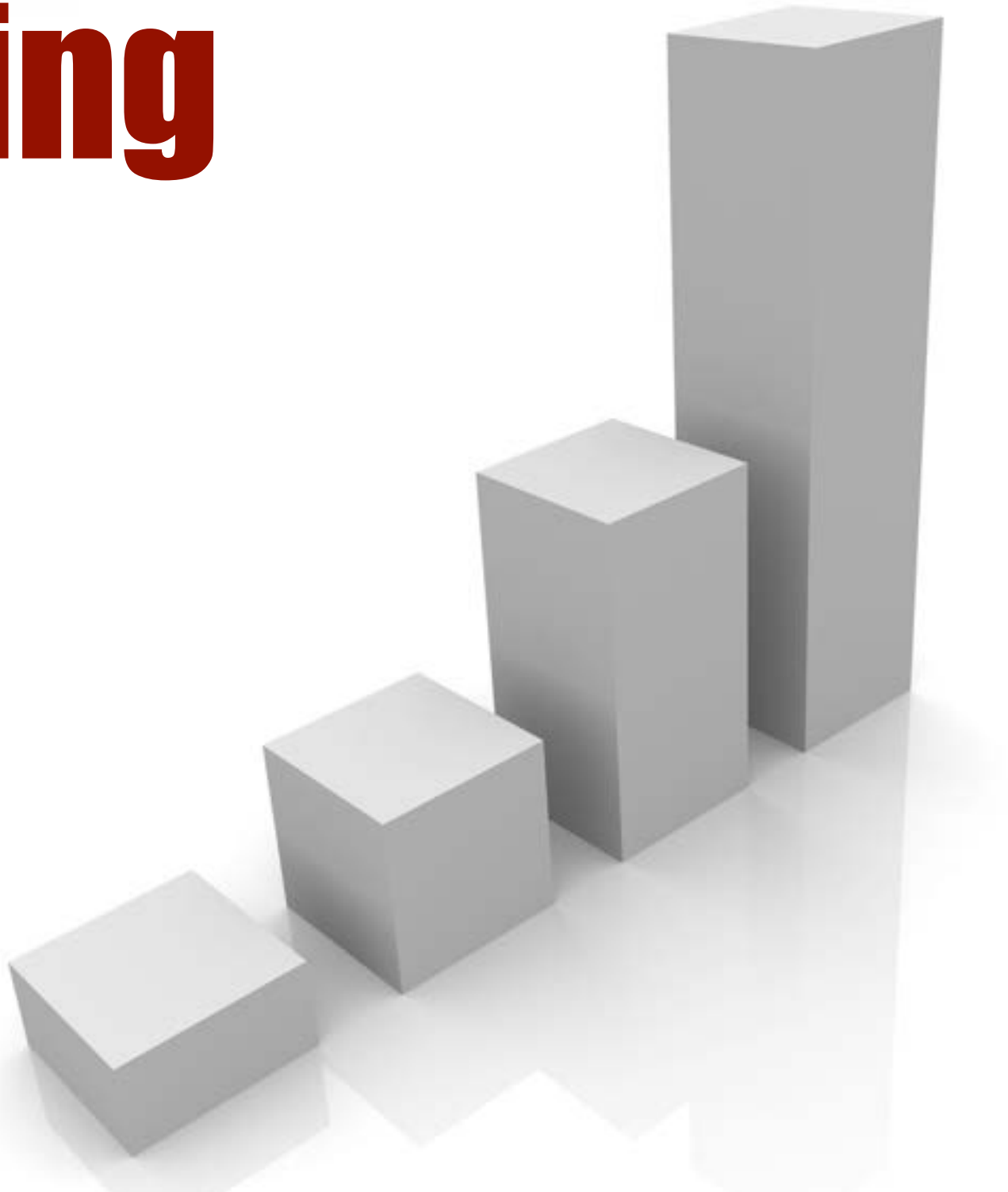
***"The process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure"***

**[Martin Fowler, 1999]**

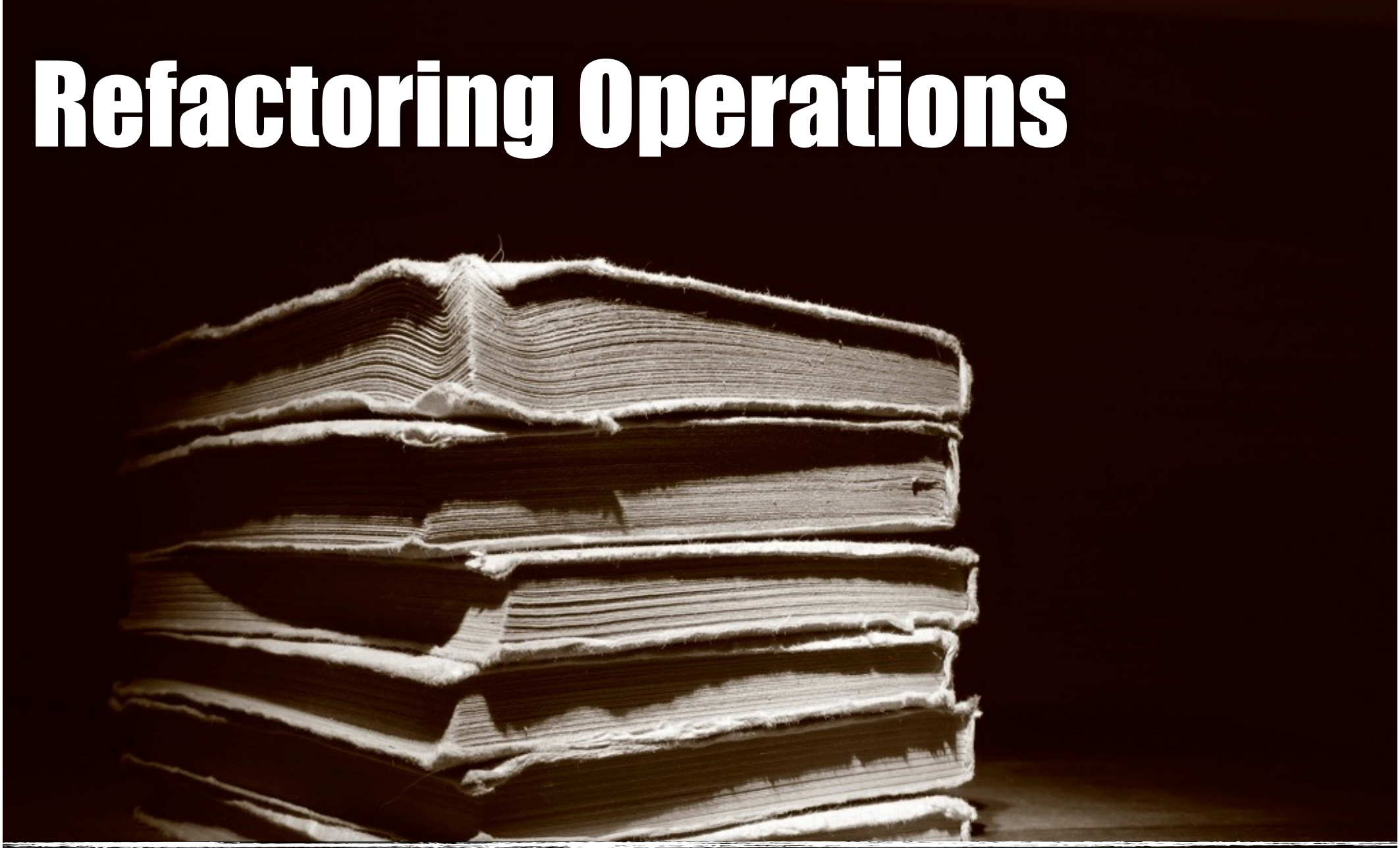


# Improving

complexity
extensibility
modularity
reusability
maintainability



# Refactoring Operations



More than 90 refactoring operations defined in the Fowler's catalog
(see http://refactoring.com/catalog/)

replace error code with exception

remove control flag

reduce scope of variable

rename field

rename method

hide method

# Most of the refactoring operations are very simple program transformations

introduce assertion

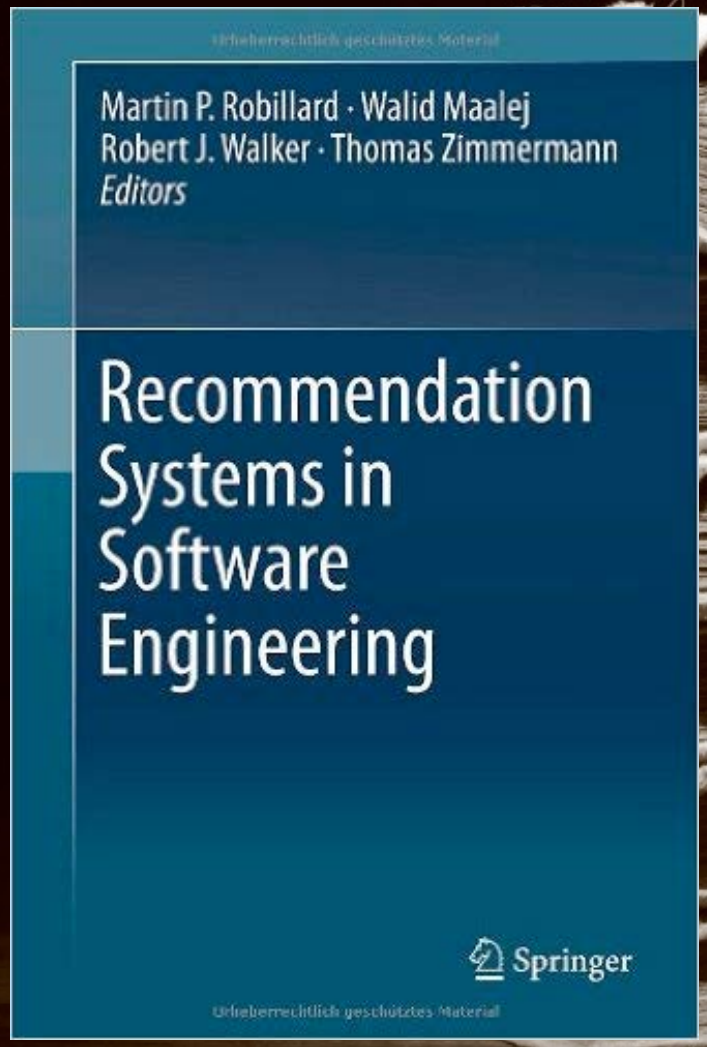
add paremeter

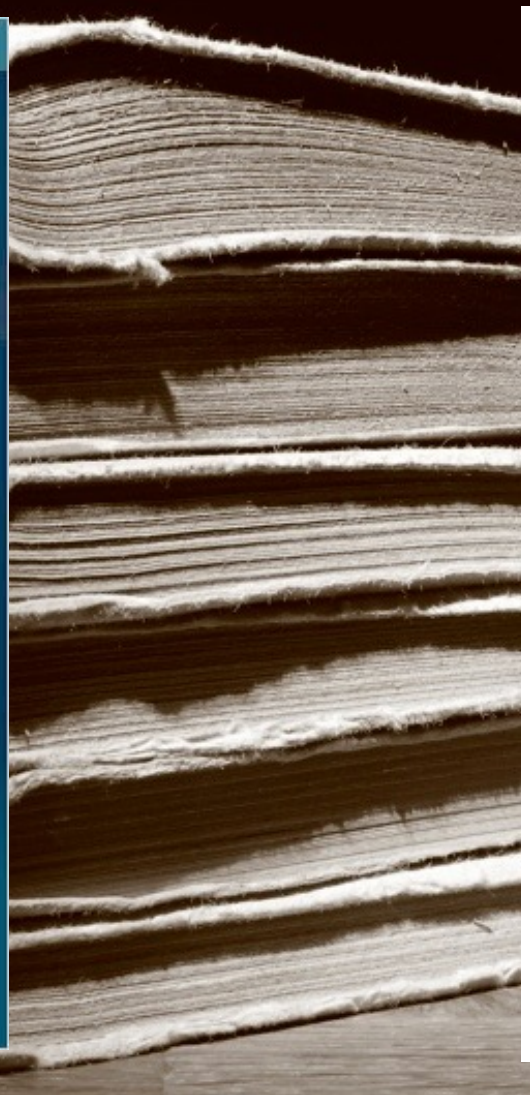
remove parameter

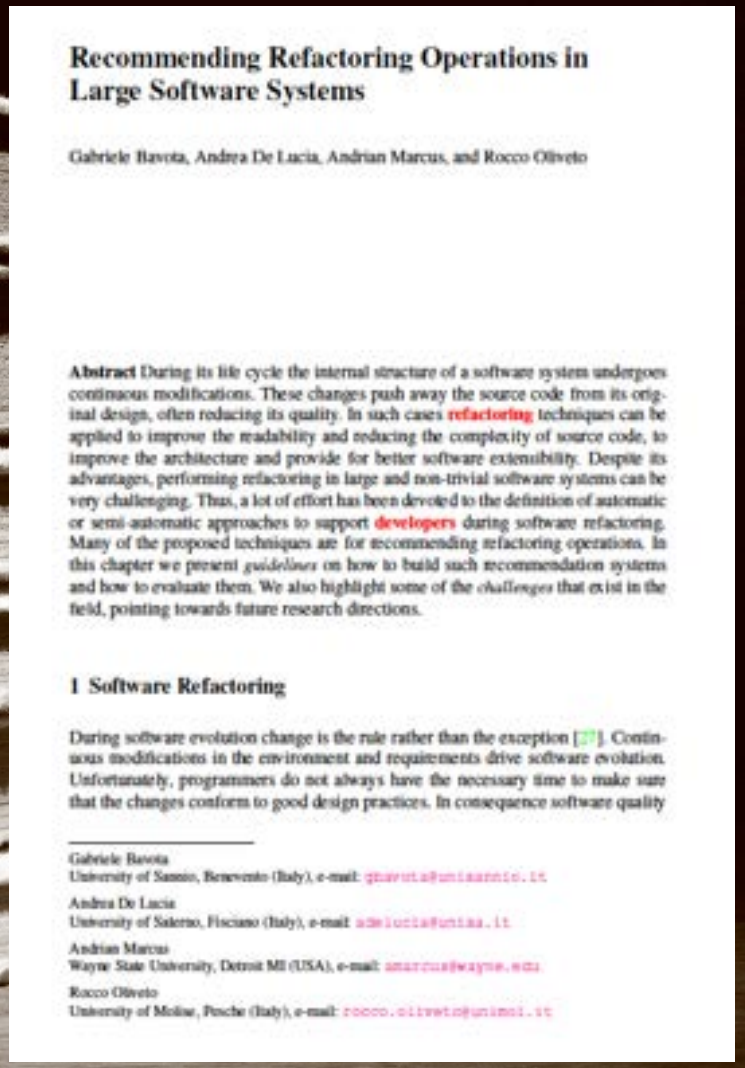
replace magic number with constant

replace exception with test

# Refactoring Operations





**Recommending Refactoring Operations in Large Software Systems**

Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto

**Abstract** During its life cycle the internal structure of a software system undergoes continuous modifications. These changes push away the source code from its original design, often reducing its quality. In such cases **refactoring** techniques can be applied to improve the readability and reducing the complexity of source code, to improve the architecture and provide for better software extensibility. Despite its advantages, performing refactoring in large and non-trivial software systems can be very challenging. Thus, a lot of effort has been devoted to the definition of automatic or semi-automatic approaches to support **developers** during software refactoring. Many of the proposed techniques are for recommending refactoring operations. In this chapter we present *guidelines* on how to build such recommendation systems and how to evaluate them. We also highlight some of the *challenges* that exist in the field, pointing towards future research directions.

## 1 Software Refactoring

During software evolution change is the rule rather than the exception [27]. Continuous modifications in the environment and requirements drive software evolution. Unfortunately, programmers do not always have the necessary time to make sure that the changes conform to good design practices. In consequence software quality

Gabriele Bavota
University of Sannio, Benevento (Italy), e-mail: gbavota@unisannio.it
Andrea De Lucia
University of Salerno, Fisciano (Italy), e-mail: adelucia@unisa.it
Andrian Marcus
Wayne State University, Detroit MI (USA), e-mail: amarcus@wayne.edu
Rocco Oliveto
University of Molise, Pesche (Italy), e-mail: rocco.oliveto@unimol.it

## Breaking code in more logical pieces

Extract Class Refactoring

Extract Package Refactoring

Extract Method Refactoring


## Improving location of code

Move Method Refactoring

Move Class Refactoring

## Improving Adherence to Object-Oriented Programming Principles

Replace Conditional With Polymorphism

Encapsulate Field

Pull Up Method (Field)

Push Down Method (Field)

## Breaking code in more logical pieces

Extract Class Refactoring

Extract Package Refactoring

Extract Method Refactoring


## Improving location of code

Move Method Refactoring

Move Class Refactoring

## Improving Adherence to Object-Oriented Programming Principles

Replace Conditional With Polymorphism

Encapsulate Field

Pull Up Method (Field)

Push Down Method (Field)


# Principles Driving Refactoring

# Cohesion

*"How strongly related and focused the various responsibilites of a software module are"*

# Cohesion

**High** cohesion is desirable

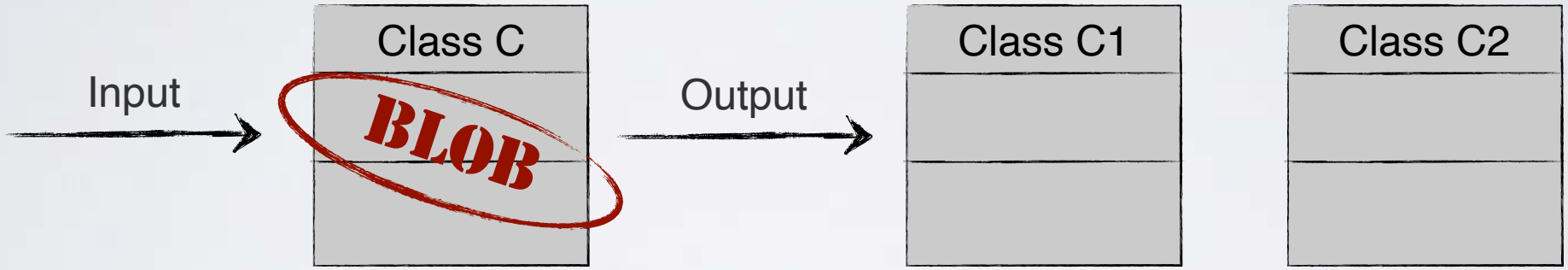
# Coupling

*"The degree to which each software module relies on each one of the other modules"*

# Coupling

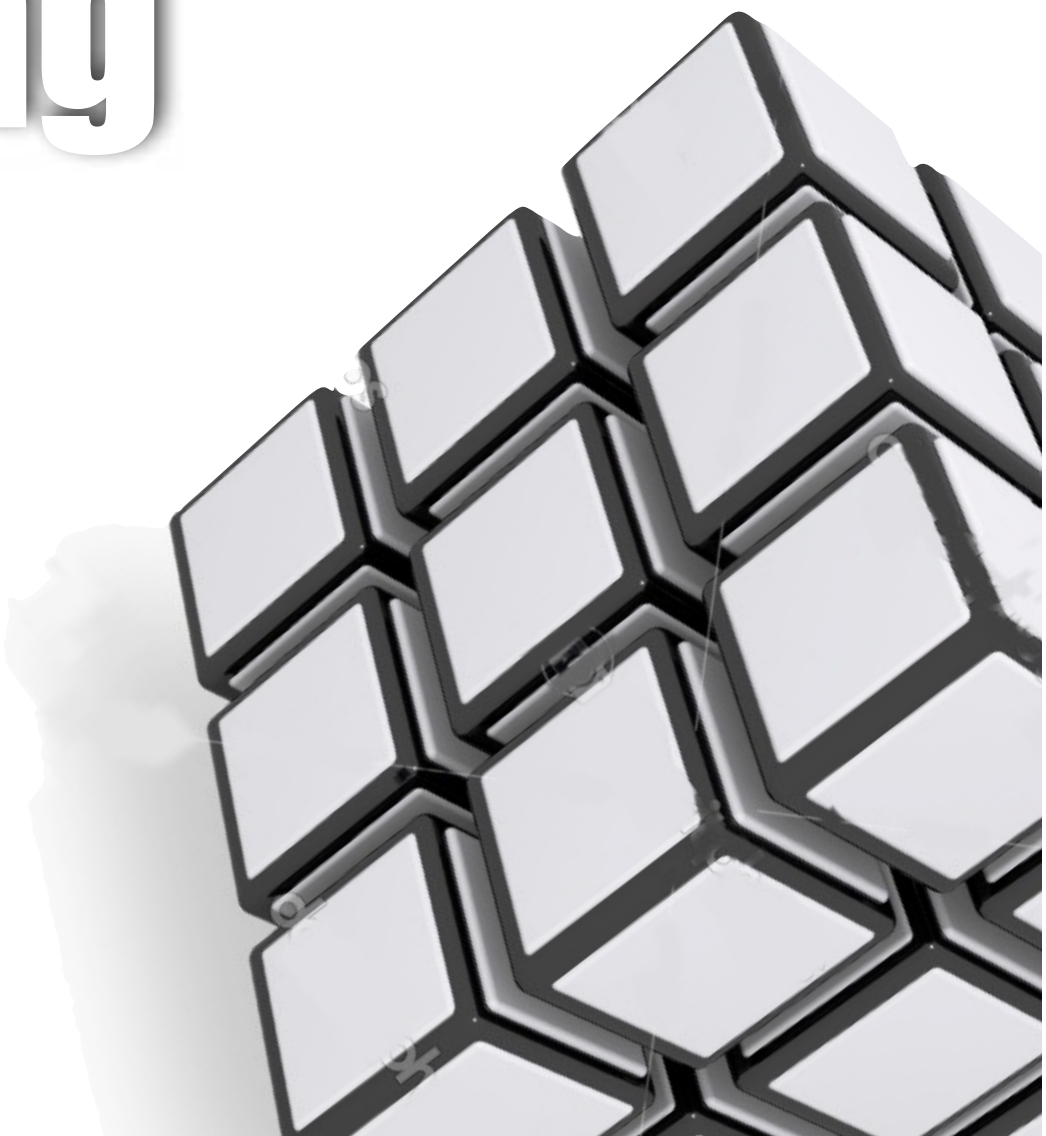
## Low coupling is desirable

# Extract Class Refactoring

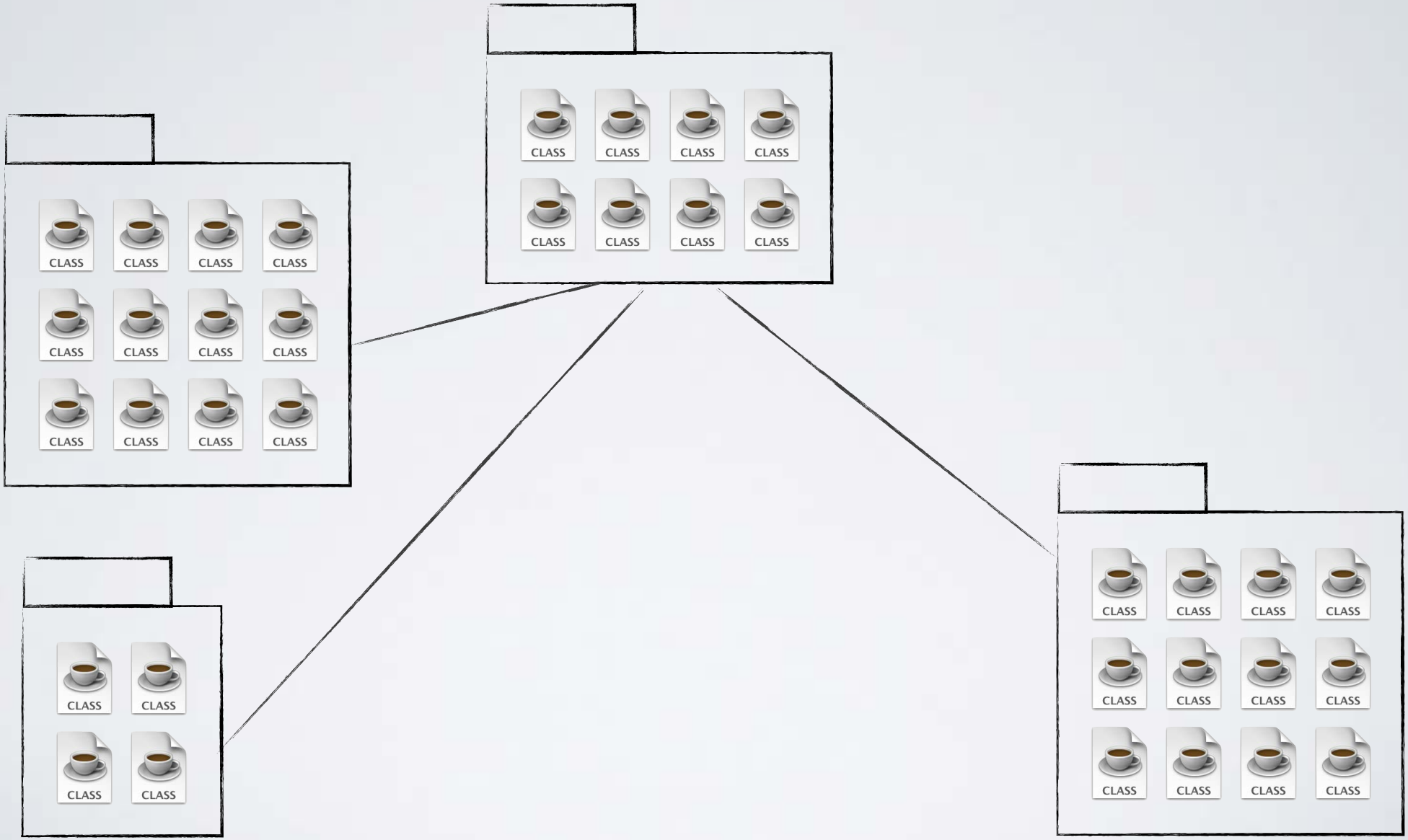



**Splitting a class with many responsibilities into different classes**

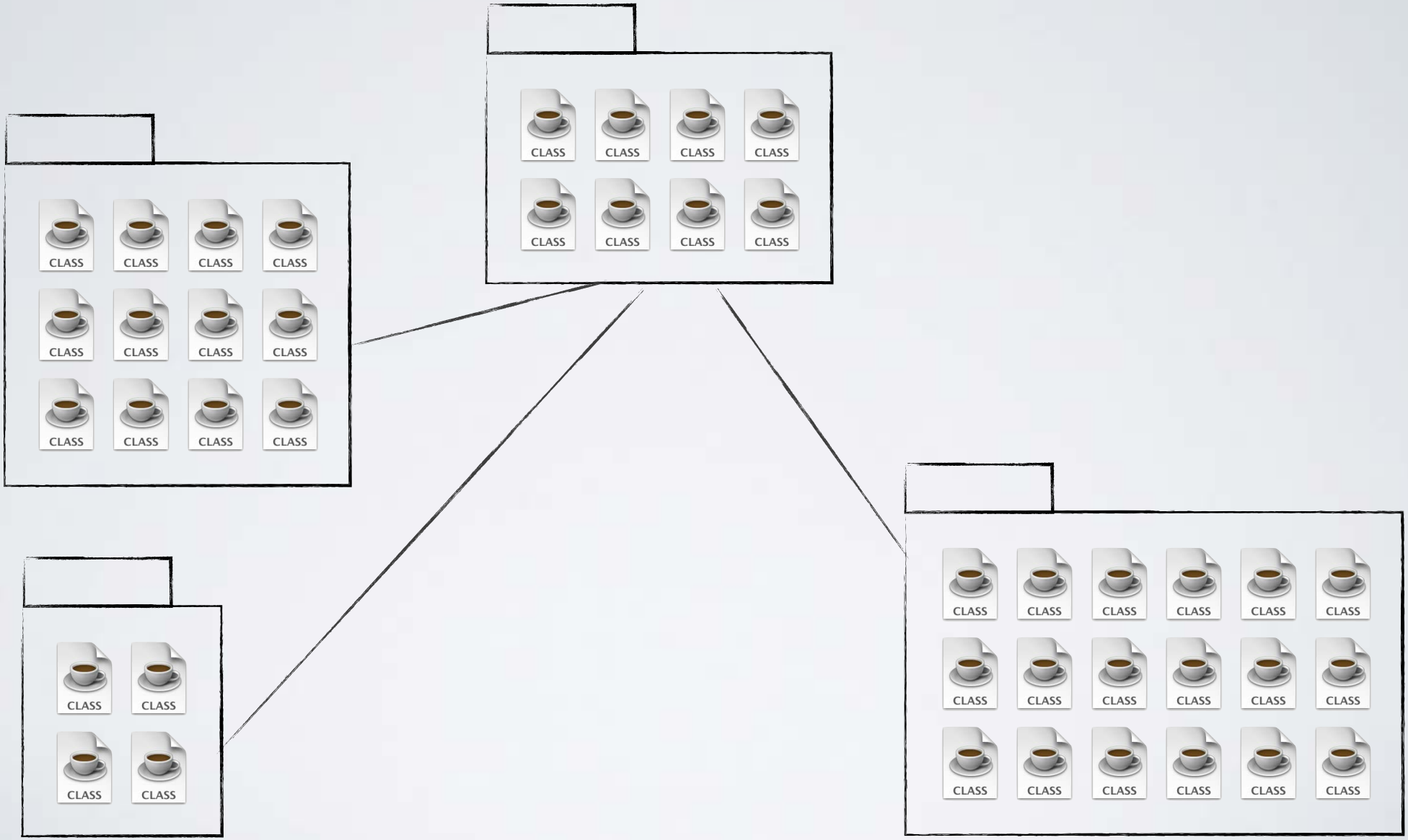
# Extract Package Refactoring



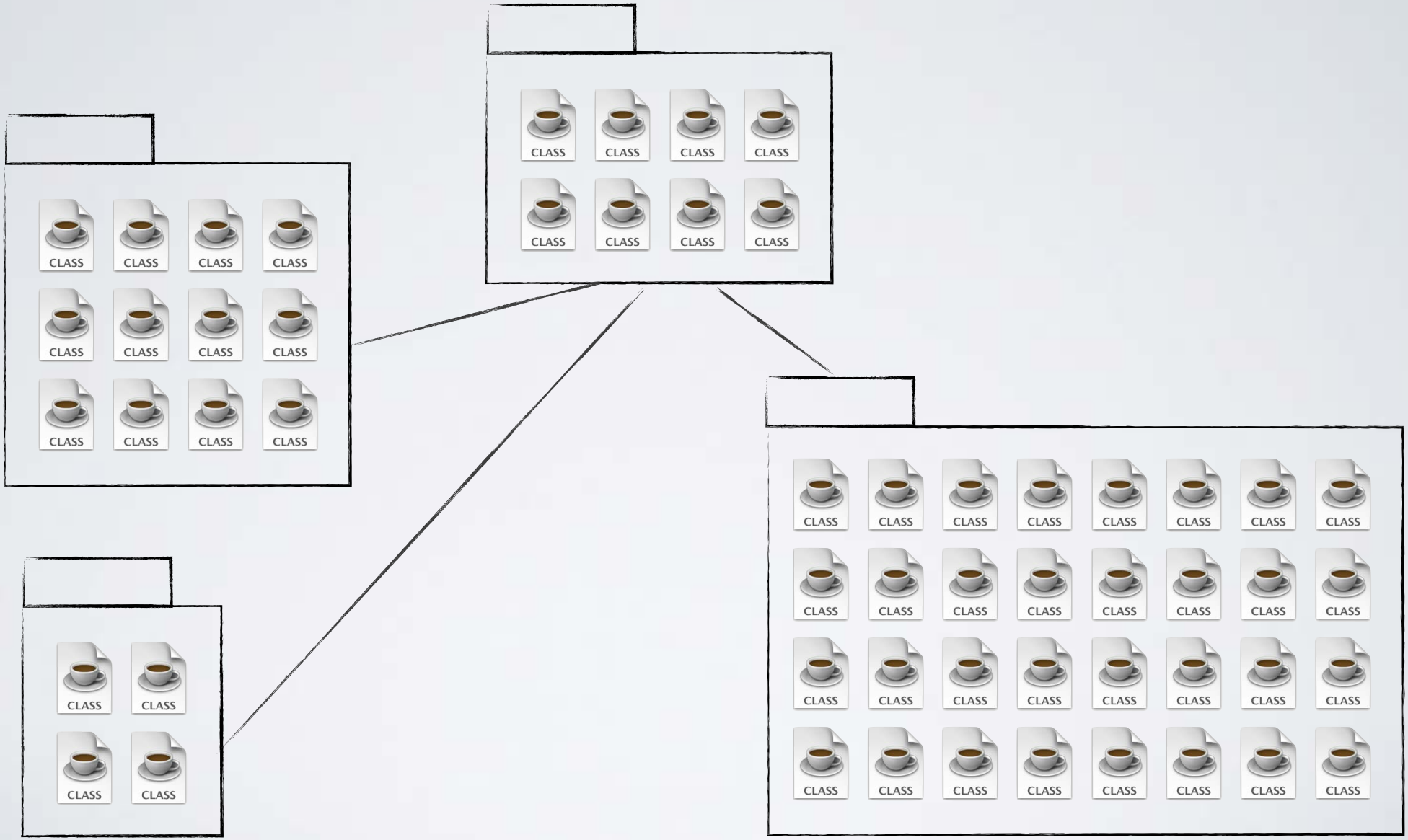
# Promiscuous packages



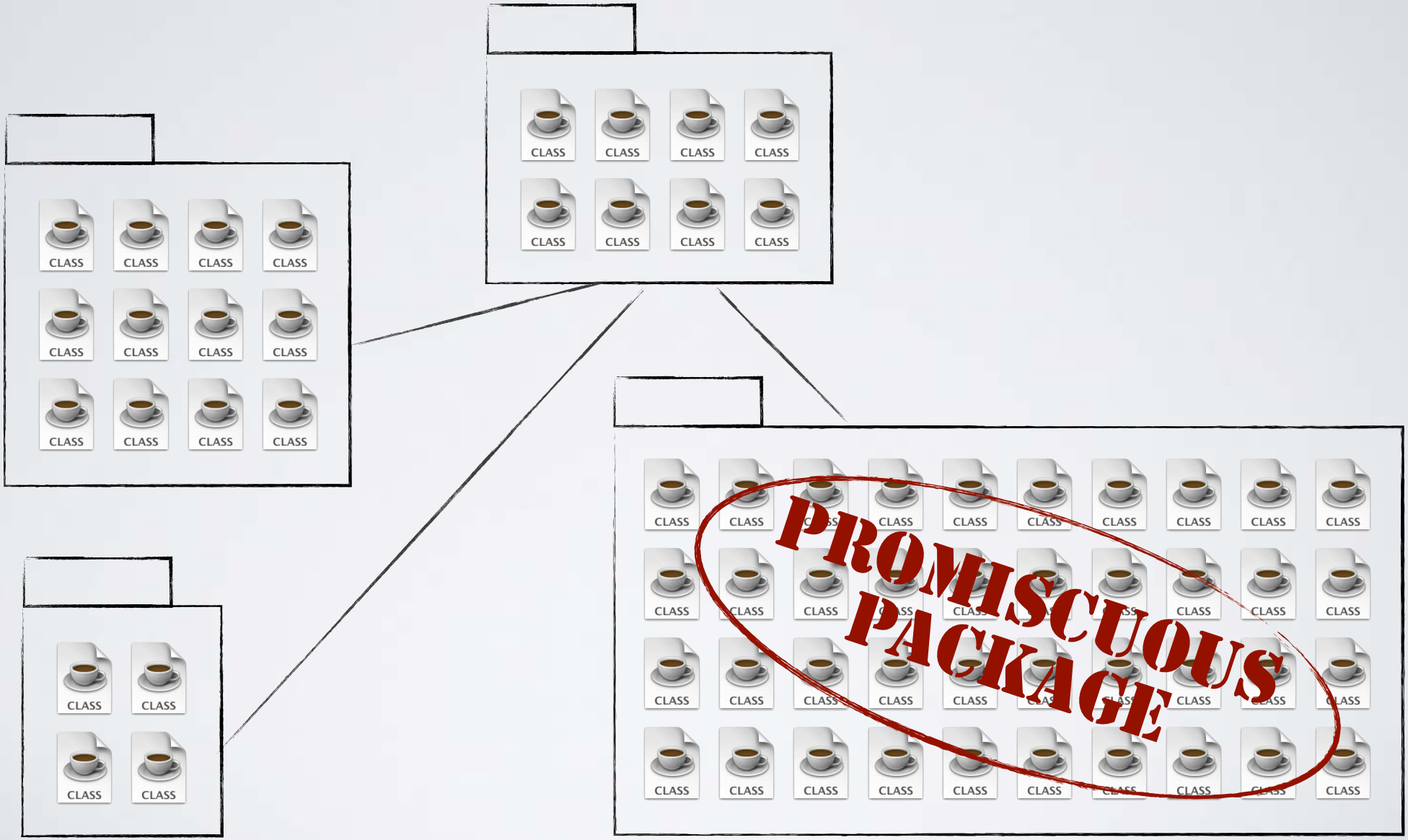
# Promiscuous packages



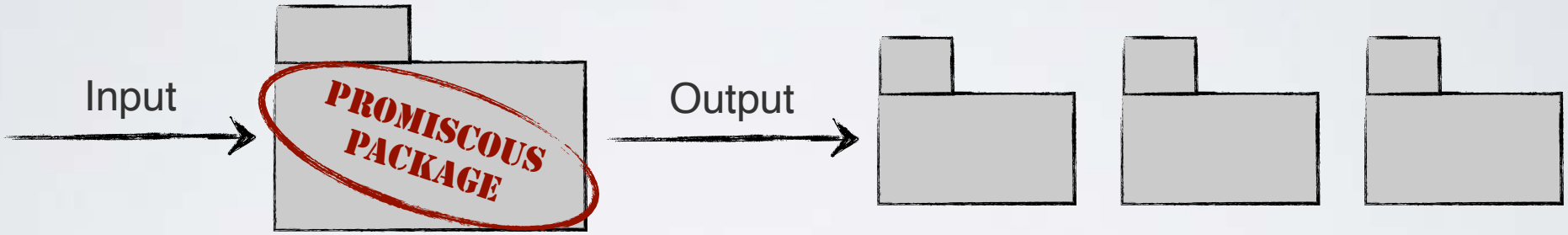
# Promiscuous packages



# Promiscuous packages



# Extract Package Refactoring




**Splitting a package grouping together classes having different responsibilities into new packages representing a well defined set of responsibilities**

# Move Method Refactoring

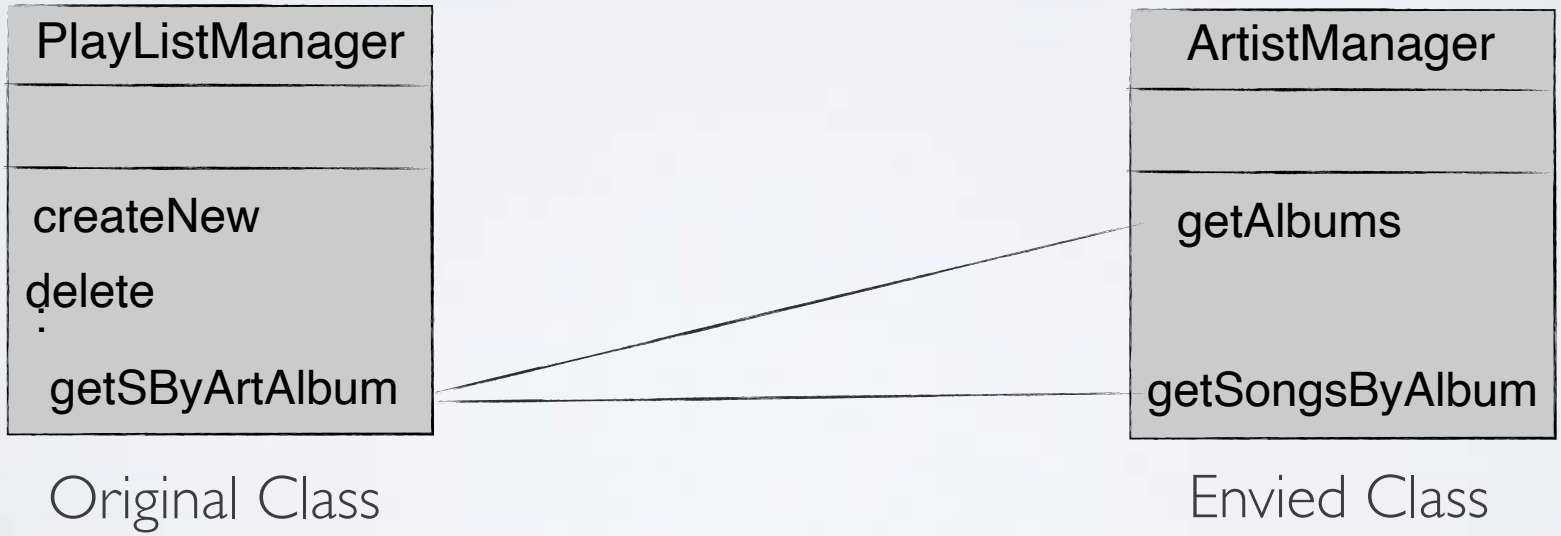
# Feature Envy Bad Smell

**"a method is more interested in a class other than the one it actually is in"**


**aTunes example**

# Move Method Refactoring

## Move the method affected by feature envy to the envied class



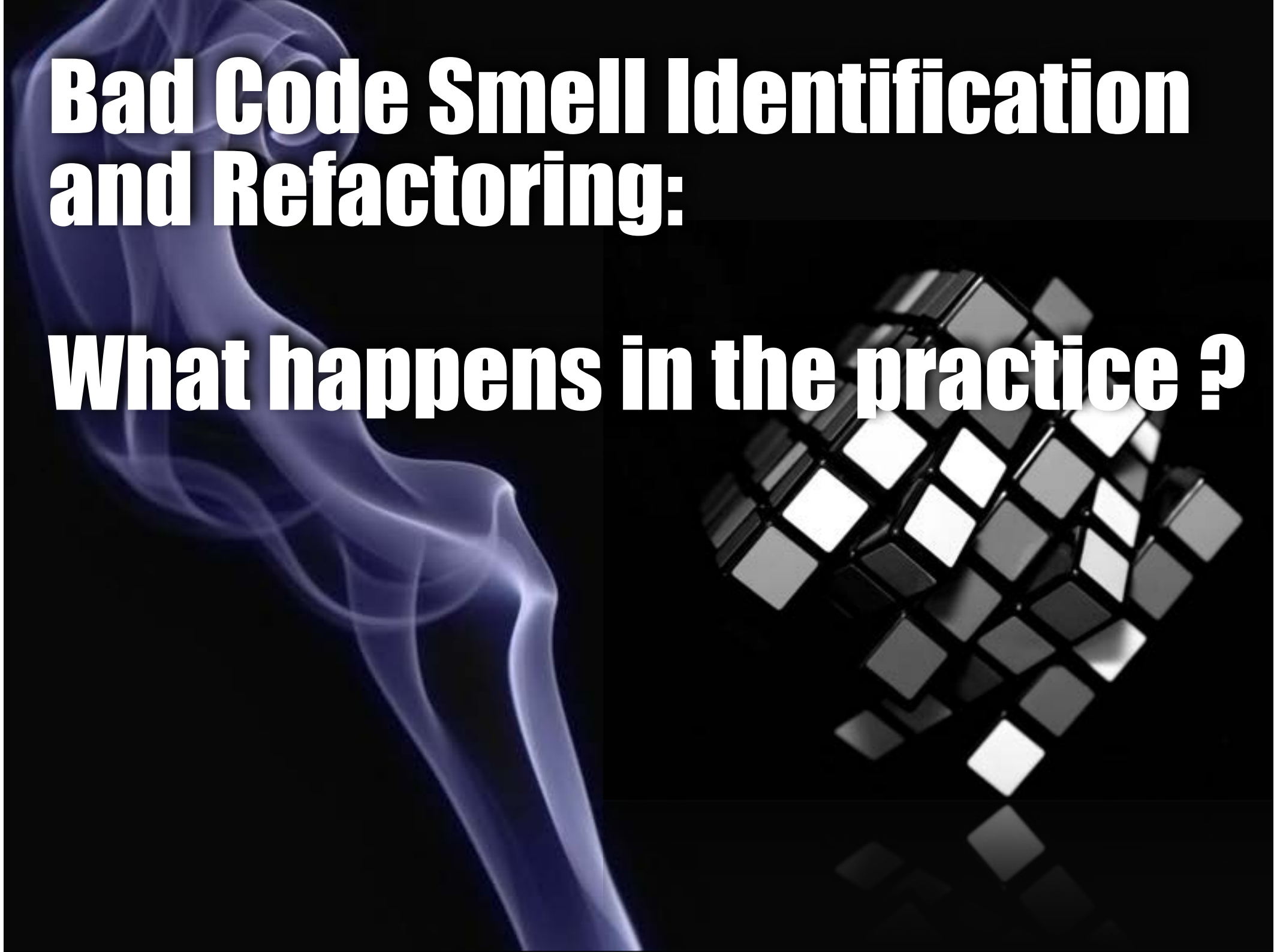
# Move Class Refactoring

One of the main reasons for architectural erosion is inconsistent placement of source code classes in software packages
[Fowler 1999]

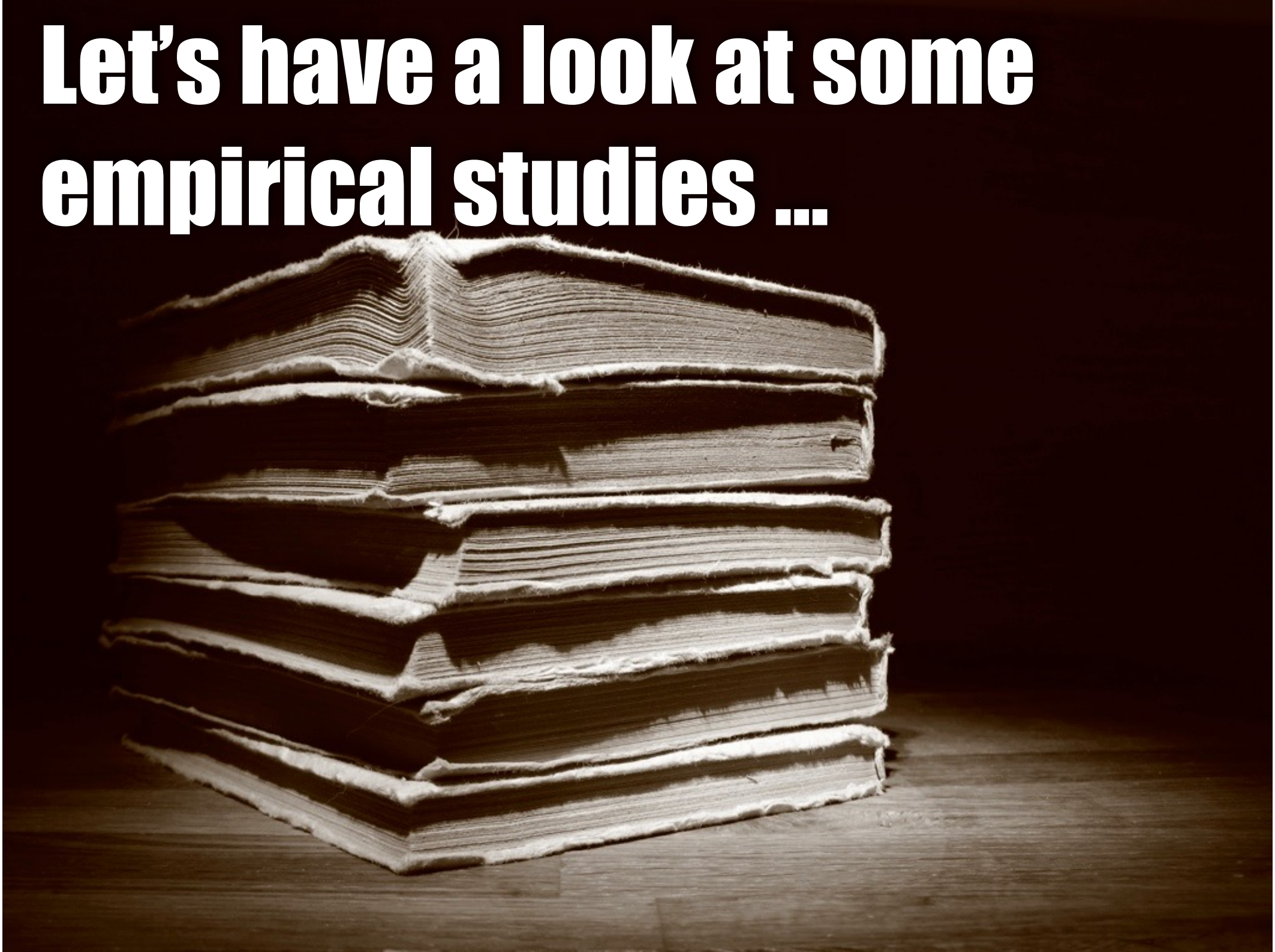
Fowler M.
*Refactoring: improving the design of existing code*
**Addison-Wesley**

# Bad Code Smell Identification and Refactoring:

# What happens in the practice ?



# Let's have a look at some empirical studies ...



# When and why your code starts to smell bad ?


[Tufano et al. - ICSE 2015]

# 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 complextity**

## 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

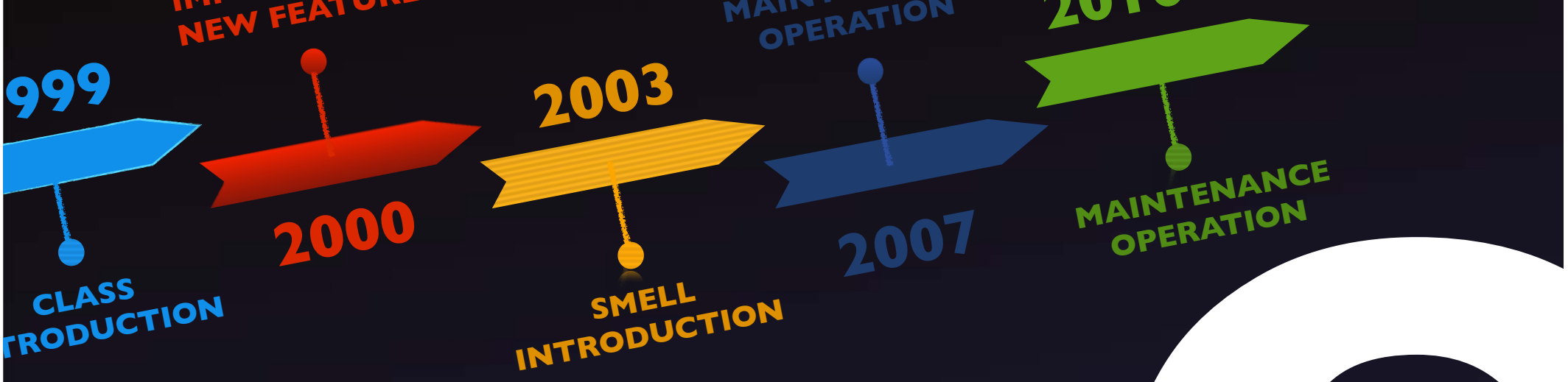


different ecosystems analyzed

# Study Design

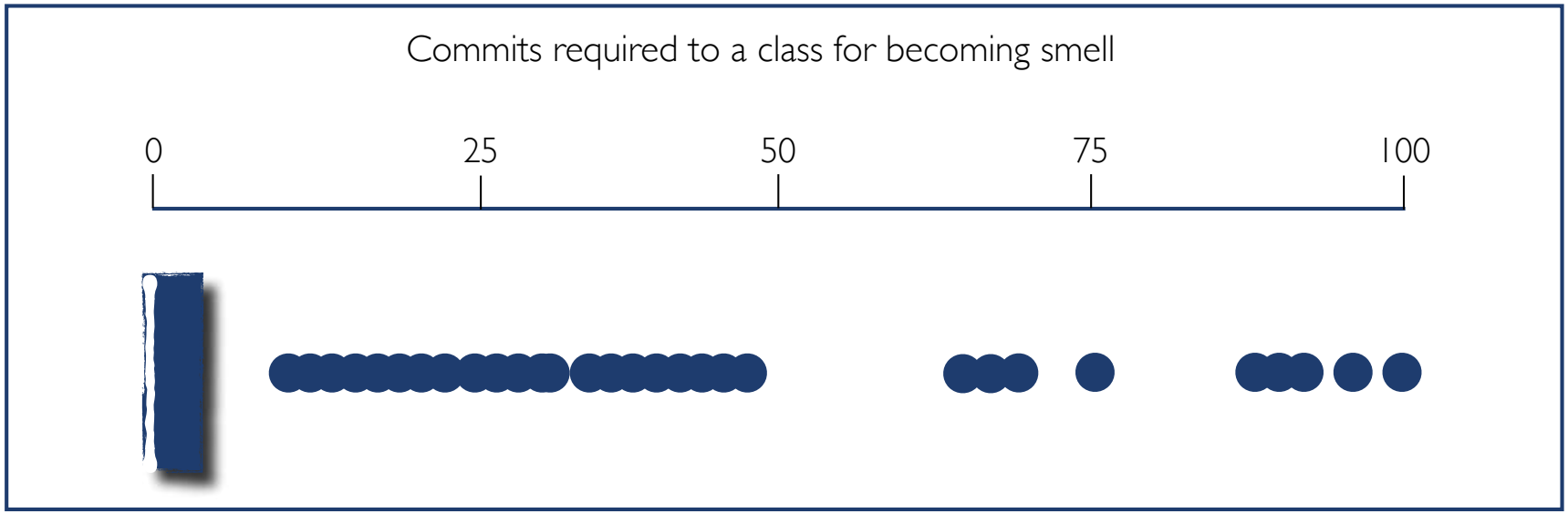
# 200

total analyzed systems



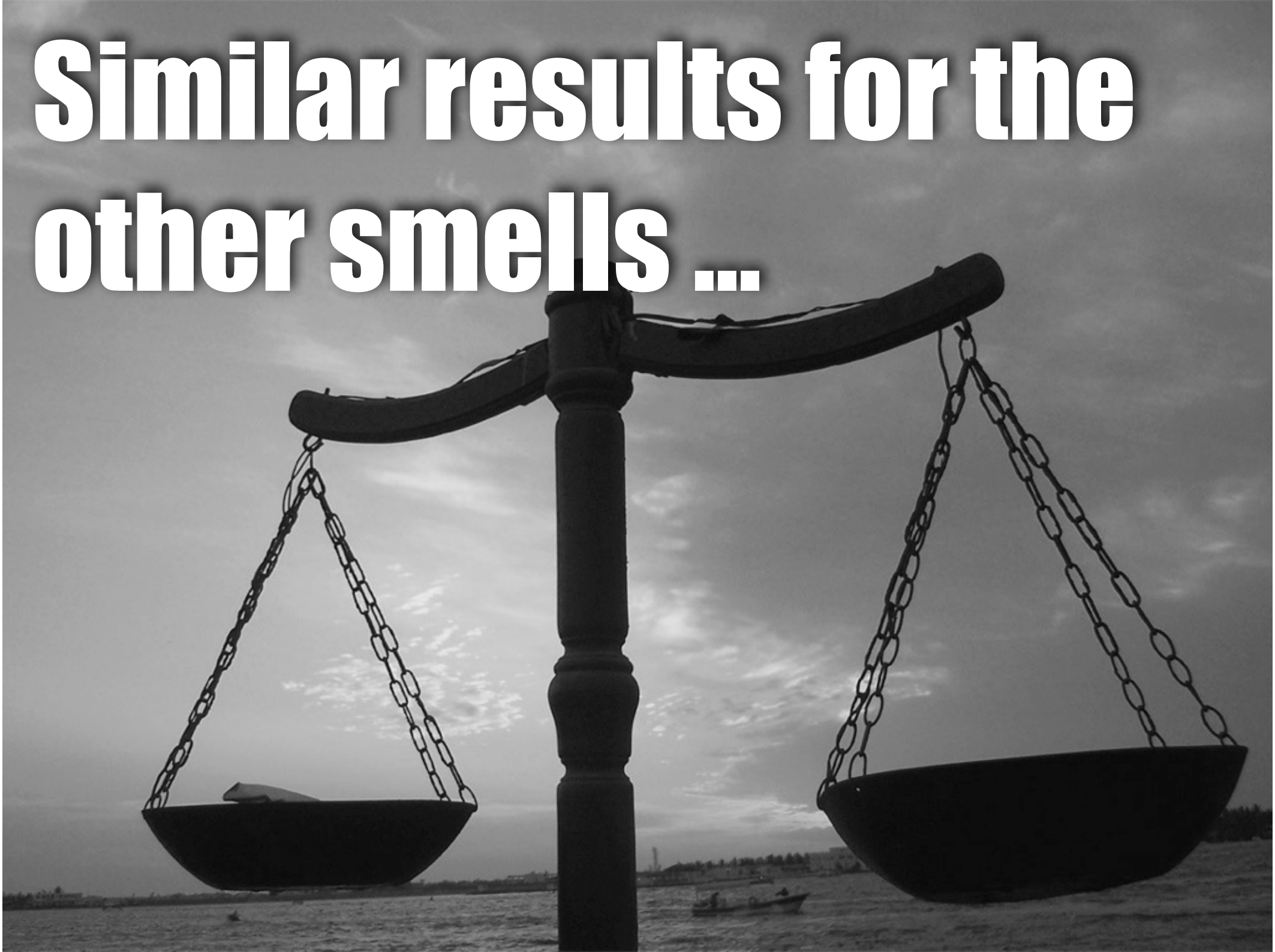

# When are code smells introduced

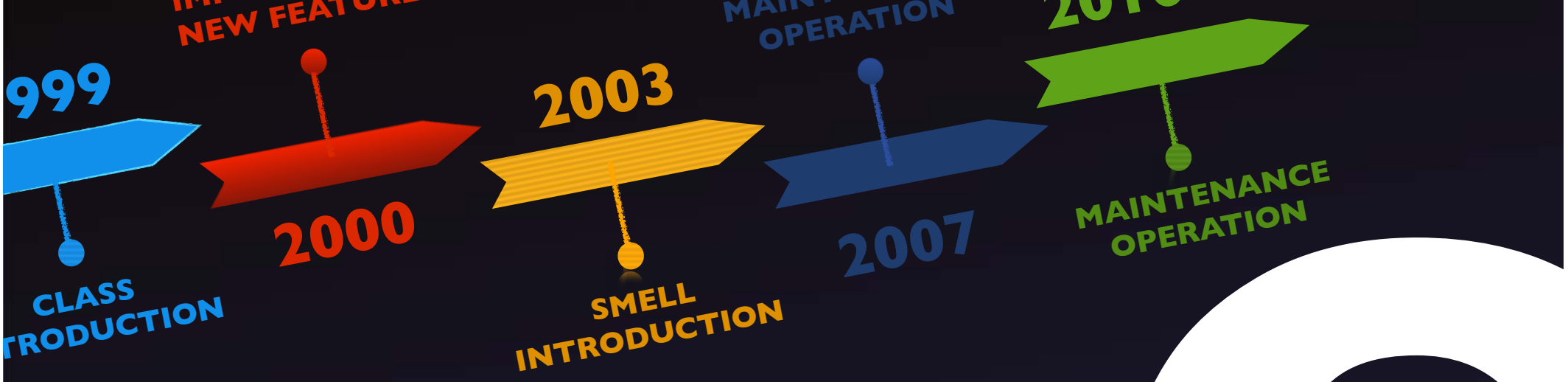
# WHEN blobs are introduced




Generally, blobs affect a class since its creation

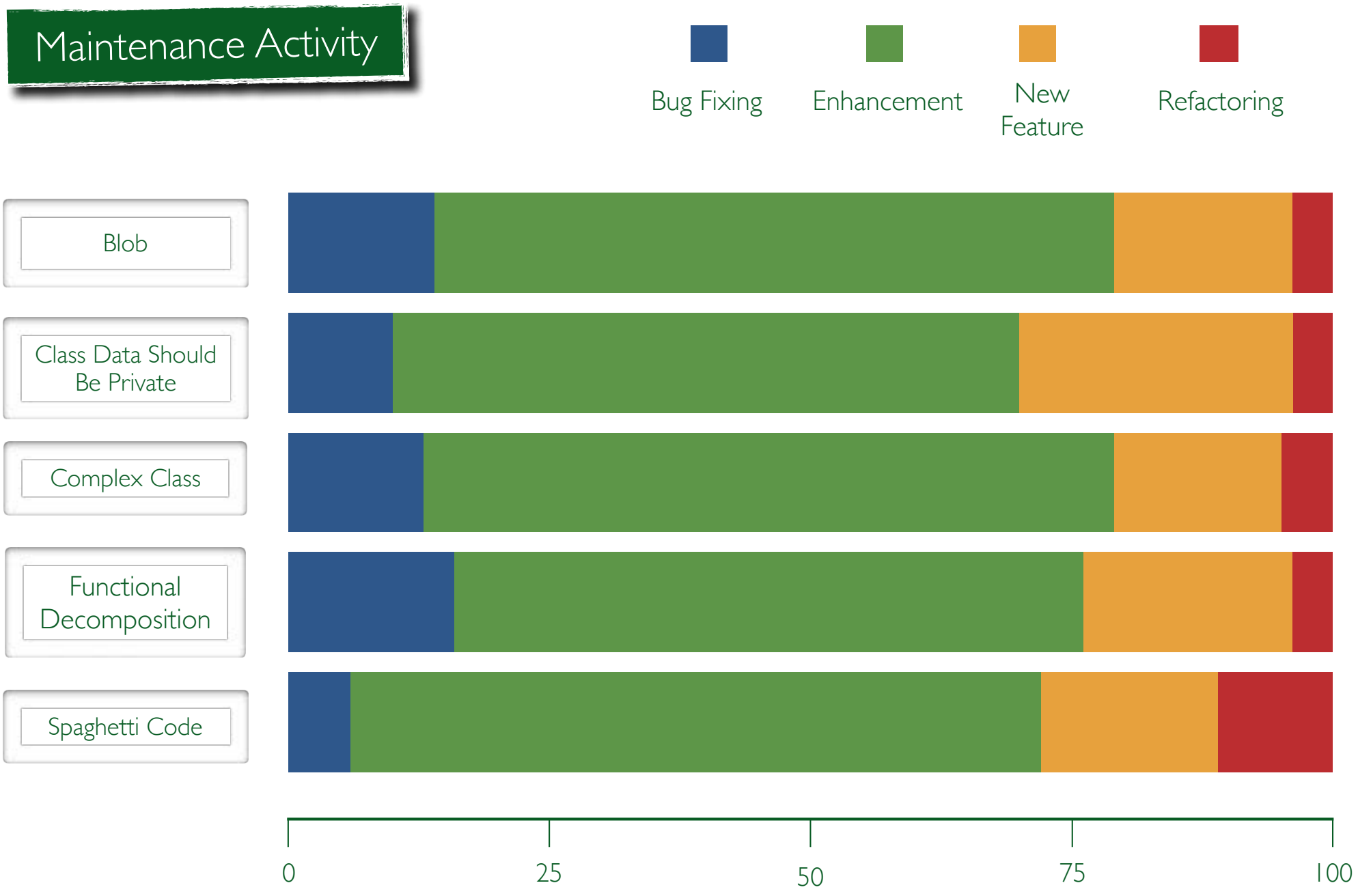
There are several cases in which a blob is introduced during class maintenance


Similar results for the other smells ...



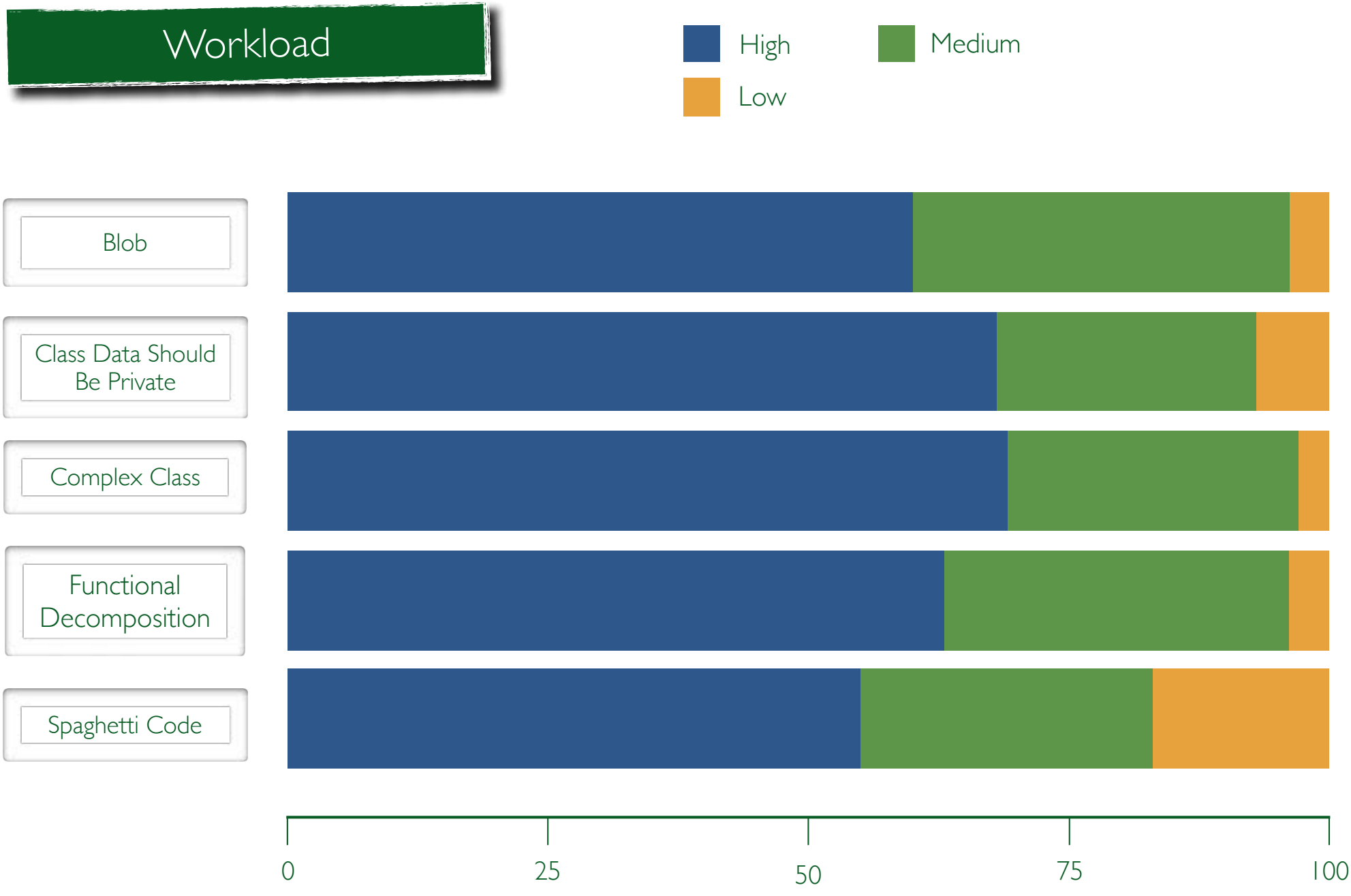

# Why are code smells introduced

# WHY are code smells introduced



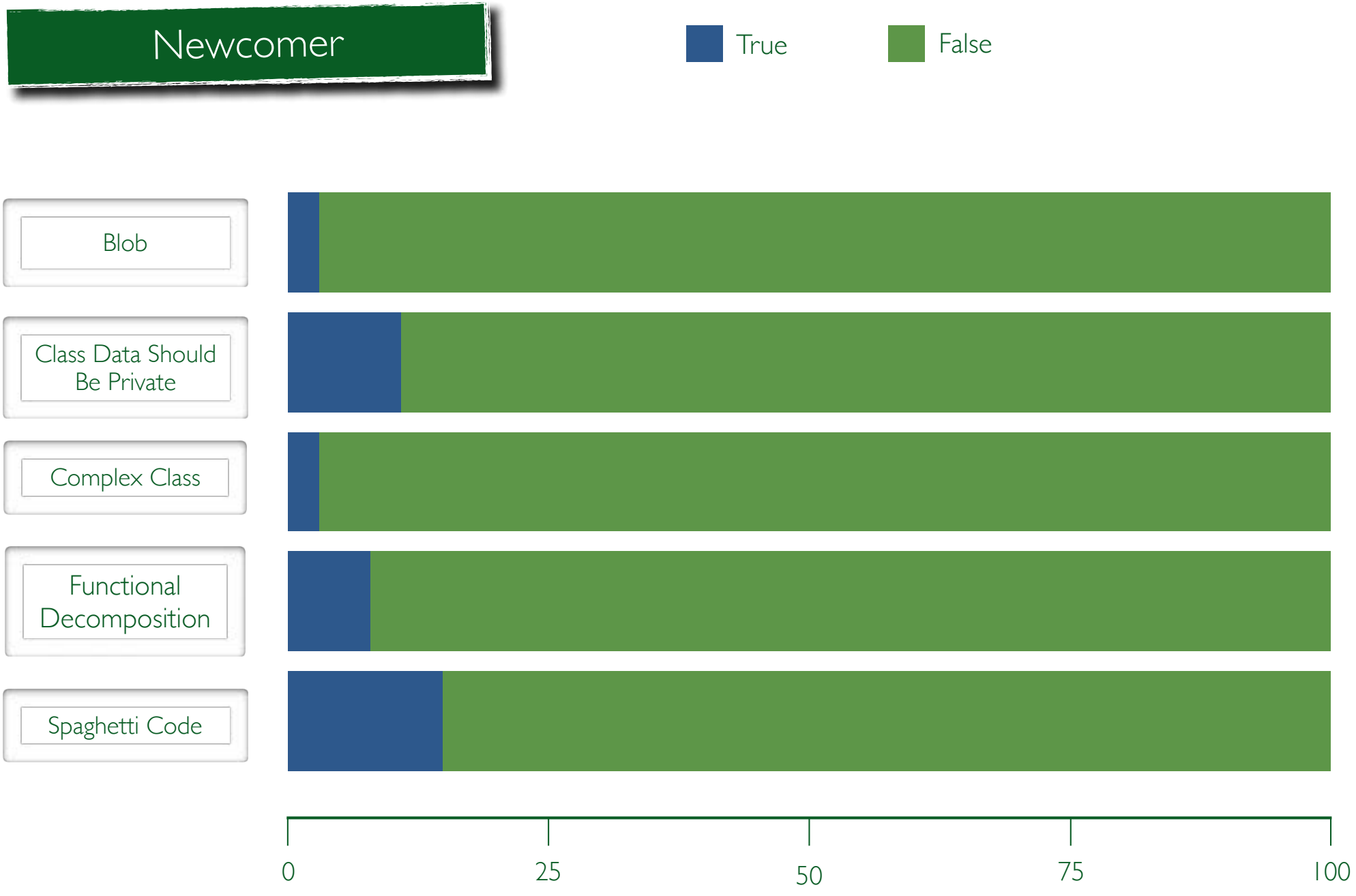
# WHY are code smells introduced

Workload

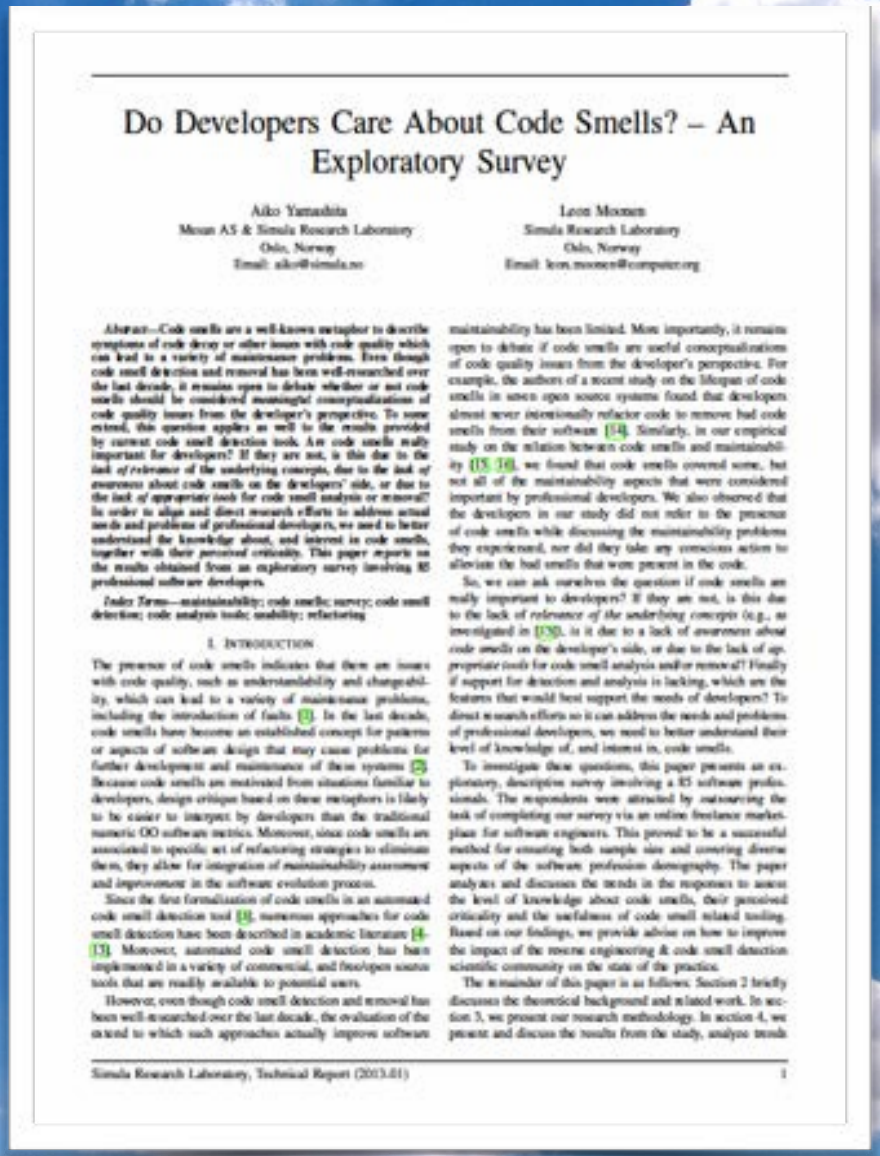


# WHY are code smells introduced

Newcomer



# How about the Developers' Perception of Bad Code Smells ?



Do Developers Care About Code Smells? – An Exploratory Survey

Aiko Yamashita
Moan AS & Simula Research Laboratory
Oslo, Norway
Email: aiko@simula.no

Leon Moonen
Simula Research Laboratory
Oslo, Norway
Email: leon.moonen@computer.org

Abstract—Code smells are a well-known metaphor to describe symptoms of code decay or other issues with code quality which can lead to a variety of maintenance problems. Even though code smell detection and removal has been well-researched over the last decade, it remains open to debate whether or not code smells should be considered *meaningful* conceptualizations of code quality issues from the developer's perspective. To some extend, this question applies as well to the results provided by current code smell detection tools. Are code smells really important for developers? If they are not, is this due to the *lack of relevance* of the underlying concepts, due to the *lack of awareness* about code smells on the developers' side, or due to the *lack of appropriate tools* for code smell analysis or removal? In order to align and direct research efforts to address actual needs and problems of professional developers, we need to better understand the knowledge about, and interest in code smells, together with their *perceived criticality*. This paper *reports* on the results obtained from an exploratory survey involving 85 professional software developers.

*Index Terms*—maintainability; code smells; survey; code smell detection; code analysis tools; usability; refactoring

I. Introduction

The presence of code smells indicates that there are issues with code quality, such as understandability and changeability, which can lead to a variety of maintenance problems, including the introduction of faults [1]. In the last decade, code smells have become an established concept for patterns or aspects of software design that may cause problems for further development and maintenance of these systems [2]. Because code smells are motivated from situations familiar to developers, design critique based on these metaphors is likely to be easier to interpret by developers than the traditional numeric OO software metrics. Moreover, since code smells are associated to specific set of refactoring strategies to eliminate them, they allow for integration of *maintainability assessment* and *improvement* in the software evolution process.

Since the first formalization of code smells in an automated code smell detection tool [3], numerous approaches for code smell detection have been described in academic literature [4–13]. Moreover, automated code smell detection has been implemented in a variety of commercial, and free/open source tools that are readily available to potential users.

However, even though code smell detection and removal has been well-researched over the last decade, the evaluation of the extend to which such approaches actually improve software maintainability has been limited. More importantly, it remains open to debate if code smells are useful conceptualizations of code quality issues from the developer's perspective. For example, the authors of a recent study on the lifespan of code smells in seven open source systems found that developers almost never intentionally refactor code to remove bad code smells from their software [14]. Similarly, in our empirical study on the relation between code smells and maintainability [15, 16], we found that code smells covered some, but not all of the maintainability aspects that were considered important by professional developers. We also observed that the developers in our study did not refer to the presence of code smells while discussing the maintainability problems they experienced, nor did they take any conscious action to alleviate the bad smells that were present in the code.

So, we can ask ourselves the question if code smells are really important to developers? If they are not, is this due to the lack of *relevance of the underlying concepts* (e.g., as investigated in [15]), is it due to a lack of *awareness about code smells* on the developer's side, or due to the lack of *appropriate tools* for code smell analysis and/or removal? Finally if support for detection and analysis is lacking, which are the features that would best support the needs of developers? To direct research efforts so it can address the needs and problems of professional developers, we need to better understand their level of knowledge of, and interest in, code smells.

To investigate these questions, this paper presents an exploratory, descriptive survey involving a 85 software professionals. The respondents were attracted by outsourcing the task of completing our survey via an online freelance marketplace for software engineers. This proved to be a successful method for ensuring both sample size and covering diverse aspects of the software profession demography. The paper analyzes and discusses the trends in the responses to assess the level of knowledge about code smells, their perceived criticality and the usefulness of code smell related tooling. Based on our findings, we provide advise on how to improve the impact of the reverse engineering & code smell detection scientific community on the state of the practice.

The remainder of this paper is as follows: Section 2 briefly discusses the theoretical background and related work. In section 3, we present our research methodology. In section 4, we present and discuss the results from the study, analyze trends



**A Survey with 85 developers, investigating:**

- **Knowledge about code smells**
- **Perceived criticality**
- **Usefulness of tools**

**Yamashita and Moonen (WCRE 2013)**

Do Developers Care About Code Smells? – An Exploratory Survey

Aiko Yamashita
Moan AS & Simula Research Laboratory
Oslo, Norway
Email: aiko@simula.no

Leon Moonen
Simula Research Laboratory
Oslo, Norway
Email: leon.moonen@computer.org

*Abstract*—Code smells are a well-known metaphor to describe symptoms of code decay or other issues with code quality which can lead to a variety of maintenance problems. Even though code smell detection and removal has been well-researched over the last decade, it remains open to debate whether or not code smells should be considered *meaningful* conceptualizations of code quality issues from the developer's perspective. To some extend, this question applies as well to the results provided by current code smell detection tools. Are code smells really important for developers? If they are not, is this due to the *lack of relevance* of the underlying concepts, due to the *lack of awareness* about code smells on the developers' side, or due to the *lack of appropriate tools* for code smell analysis or removal? In order to align and direct research efforts to address actual needs and problems of professional developers, we need to better understand the knowledge about, and interest in code smells, together with their *perceived criticality*. This paper *reports* on the results obtained from an exploratory survey involving 85 professional software developers.

*Index Terms*—maintainability; code smells; survey; code smell detection; code analysis tools; usability; refactoring

## I. Introduction

The presence of code smells indicates that there are issues with code quality, such as understandability and changeability, which can lead to a variety of maintenance problems, including the introduction of faults [1]. In the last decade, code smells have become an established concept for patterns or aspects of software design that may cause problems for further development and maintenance of these systems [2]. Because code smells are motivated from situations familiar to developers, design critique based on these metaphors is likely to be easier to interpret by developers than the traditional numeric OO software metrics. Moreover, since code smells are associated to specific set of refactoring strategies to eliminate them, they allow for integration of *maintainability assessment* and *improvement* in the software evolution process.

Since the first formalization of code smells in an automated code smell detection tool [3], numerous approaches for code smell detection have been described in academic literature [4–13]. Moreover, automated code smell detection has been implemented in a variety of commercial, and free/open source tools that are readily available to potential users.

However, even though code smell detection and removal has been well-researched over the last decade, the evaluation of the extend to which such approaches actually improve software maintainability has been limited. More importantly, it remains open to debate if code smells are useful conceptualizations of code quality issues from the developer's perspective. For example, the authors of a recent study on the lifespan of code smells in seven open source systems found that developers almost never intentionally refactor code to remove bad code smells from their software [14]. Similarly, in our empirical study on the relation between code smells and maintainability [15, 16], we found that code smells covered some, but not all of the maintainability aspects that were considered important by professional developers. We also observed that the developers in our study did not refer to the presence of code smells while discussing the maintainability problems they experienced, nor did they take any conscious action to alleviate the bad smells that were present in the code.

So, we can ask ourselves the question if code smells are really important to developers? If they are not, is this due to the lack of *relevance of the underlying concepts* (e.g., as investigated in [15]), is it due to a lack of *awareness about code smells* on the developer's side, or due to the lack of *appropriate tools* for code smell analysis and/or removal? Finally if support for detection and analysis is lacking, which are the features that would best support the needs of developers? To direct research efforts so it can address the needs and problems of professional developers, we need to better understand their level of knowledge of, and interest in, code smells.

To investigate these questions, this paper presents an exploratory, descriptive survey involving a 85 software professionals. The respondents were attracted by outsourcing the task of completing our survey via an online freelance marketplace for software engineers. This proved to be a successful method for ensuring both sample size and covering diverse aspects of the software profession demography. The paper analyzes and discusses the trends in the responses to assess the level of knowledge about code smells, their perceived criticality and the usefulness of code smell related tooling. Based on our findings, we provide advise on how to improve the impact of the reverse engineering & code smell detection scientific community on the state of the practice.

The remainder of this paper is as follows: Section 2 briefly discusses the theoretical background and related work. In section 3, we present our research methodology. In section 4, we present and discuss the results from the study, analyze trends



# Results

**A considerably large proportion (32%) of respondents stated that they did not know about code smells**

**The majority of respondents (with knowledge about code smells) were moderately concerned about their criticality**

**The majority of respondents expressed the need for better tools**

**Yamashita and Moonen (WCRE 2013)**

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

2014 IEEE International Conference on Software Maintenance and Evolution

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

Fabio Palomba[1], Gabriele Bavota[2], Massimiliano Di Penta[3], Rocco Oliveto[4], Andrea De Lucia[1]
[1]University of Salerno, Italy [2]University of Lugano, Italy [4]University of Molise, Italy

Palomba et al. (ICSME 2014)

# 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

## Inappropriate intimacy

**Two classes exhibiting high coupling between them**

## Lazy Class

**A very small class that does not do too much in the system**

## Long Method

**A method having a huge size.**

## Long Parameter List

**A method having a long list of parameters**

## Middle Man

**A class delegating all its work to other classes**

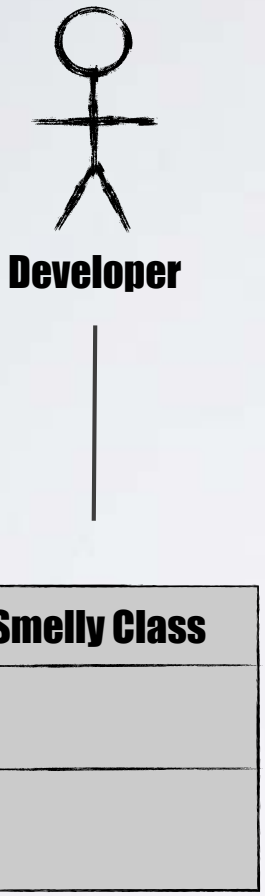
## Refused Bequest

**A class inheriting functionalities that it never uses**

## Speculative Generality

**An abstract class that is not actually needed, as it is not specialized by any other class**

# 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 a five-points Likert scale: 1 (very low), 2 (low), 3 (medium), 4 (high), 5 (very high).

# Results

# Smells generally not Perceived as Design or Implementation Problems

| | | |
|---|---|---|
| **Class Data Should Be Private** | < 25% PERCEIVED | < 24% IDENTIFIED |
| **Middle Man** | < 10% PERCEIVED | < 5% IDENTIFIED |
| **Long Parameter List** | < 36% PERCEIVED | < 20% IDENTIFIED |
| **Lazy Class** | < 20% PERCEIVED | < 5% IDENTIFIED |
| **Inappropriate Intimacy** | < 30% PERCEIVED | < 13% IDENTIFIED |

# Smells generally Perceived and Identified by Respondents

| | | |
|---|---|---|
| God Class | > 85% PERCEIVED | > 83% IDENTIFIED |
| Complex Class | > 75% PERCEIVED | > 75% IDENTIFIED |
| Long Method | > 70% PERCEIVED | > 70% IDENTIFIED |
| Spaghetti Code | > 68% PERCEIVED | < 65% IDENTIFIED |

This is consistent with study by Yamashita and Moonen (WCRE 2013)

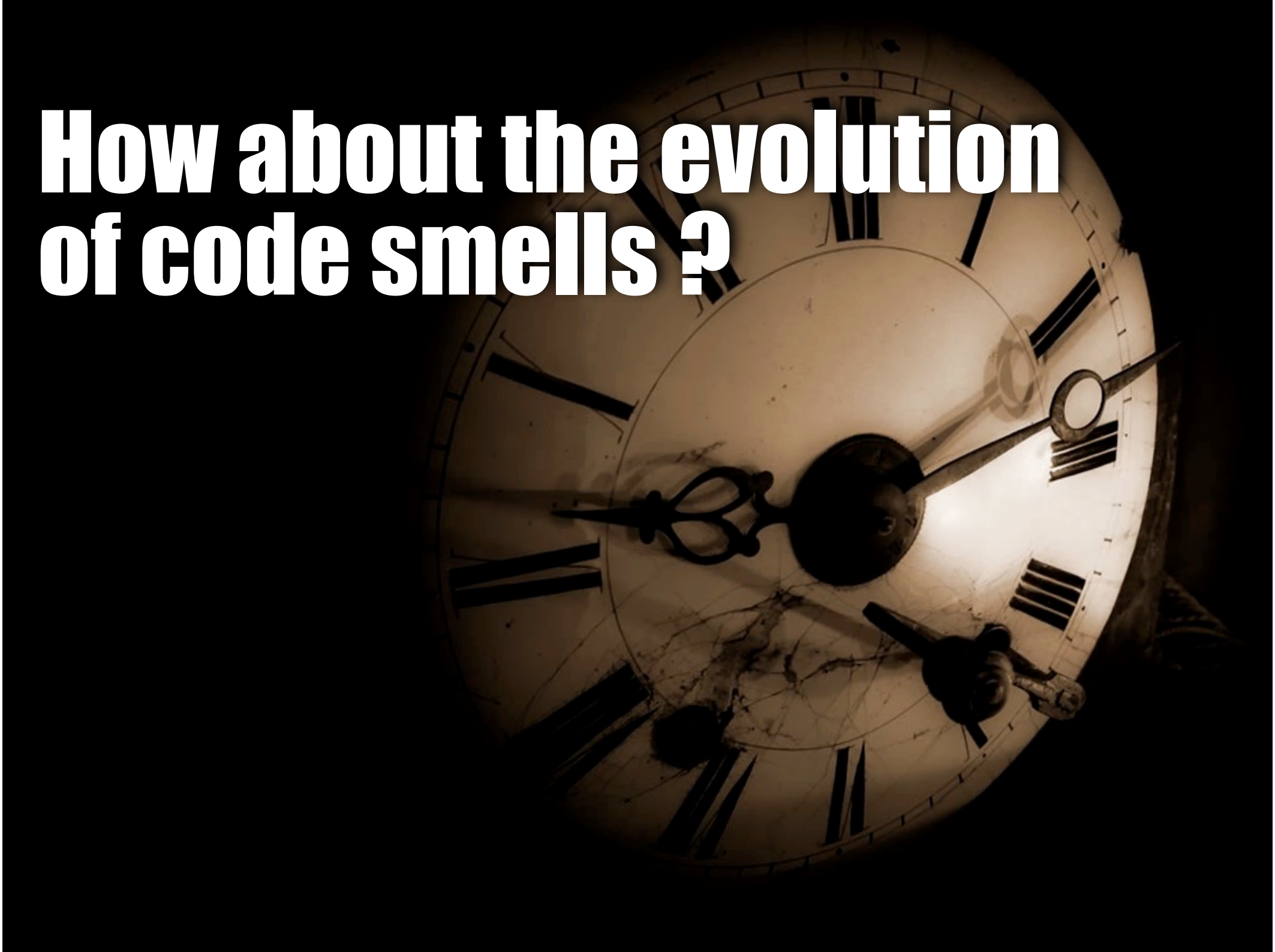
# Smells whose Perception may vary

**Refused Bequest**

**Feature Envy**

**Speculative Generality**

# How about the evolution of code smells ?



# Studies analyzing the lifespan of code smells ...

Evaluating the Lifespan of Code Smells using Software Repository Mining

Ralph Peters
*Delft University of Technology*
*The Netherlands*
*Email: ralphpeters87@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.

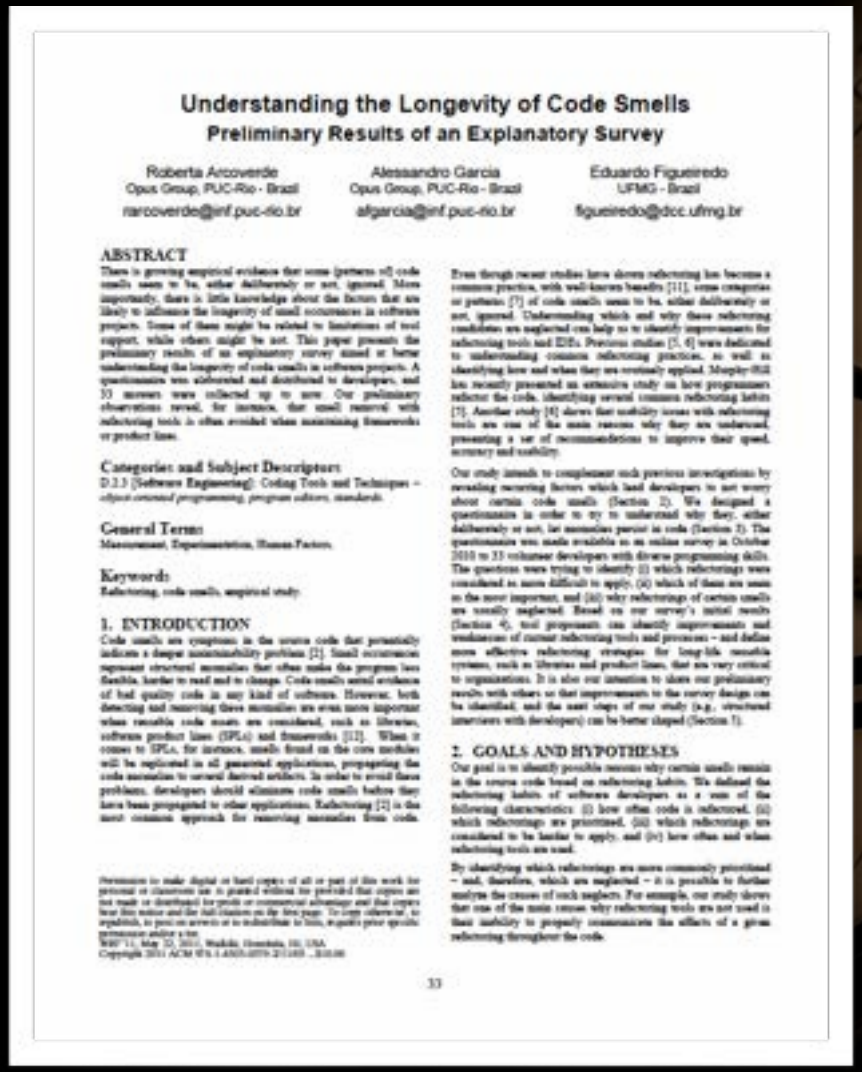
### A. Code Smells

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

**Even when developers are aware of code smells, they are not very concerned about their impact (as refactoring is rather limited) [Peters and Zaidman - CSMR 2012]**

# ... their longevity ...



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 (pattern 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 these might be related to limitations of tool support, while others might be not. This paper presents the preliminary results of an exploratory 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 send 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 conditions 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 [7]. 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 seen 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.





**Developers deliberately postpone refactorings for different reasons**

**[Arcoverde et al. - IWRT 2011]**

# ... and their evolution (1/2)

Investigating the evolution of code smells in object-oriented systems

Alexander Chatzigeorgiou · Anastasios Manakos

Received: 29 June 2011 / Accepted: 6 April 2013
© Springer-Verlag London 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 to 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: manosa912@uom.gr

Published online: 25 April 2013

## 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-

**In most 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]**

# ... and their evolution

When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away)

Michele Tufano, *Student Member, IEEE*, Fabio Palomba, *Member, IEEE*, Gabriele Bavota, *Member, IEEE*, Rocco Oliveto, *Member, IEEE*, Massimiliano Di Penta, *Member, IEEE*, Andrea De Lucia, *Senior Member, IEEE*, and Denys Poshyvanyk, *Member, IEEE*

**Abstract**—Technical debt is a metaphor introduced by Cunningham to indicate "not quite right code which we postpone making it right". One noticeable symptom of technical debt is represented by code smells, defined as symptoms of poor design and implementation choices. Previous studies showed the negative impact of code smells on the comprehensibility and maintainability of code. While the repercussions of smells on code quality have been empirically assessed, there is still only anecdotal evidence on when and why bad smells are introduced, what is their survivability, and how they are removed by developers. To empirically corroborate such anecdotal evidence, we conducted a large empirical study over the change history of 200 open source projects. This study required the development of a strategy to identify smell-introducing commits, the mining of over half a million of commits, and the manual analysis and classification of over 10K of them. Our findings mostly contradict common wisdom, showing that most of the smell instances are introduced when an artifact is created and not as a result of its evolution. At the same time, 80 percent of smells survive in the system. Also, among the 20 percent of removed instances, only 9 percent are removed as a direct consequence of refactoring operations.

**Index Terms**—Code smells, empirical study, mining software repositories

## 1 Introduction

Technical debt metaphor introduced by Cunningham [22] explains well the trade-offs between delivering the most appropriate but still immature product, in the shortest time possible [14], [22], [67], [47], [70]. Bad code smells (shortly "code smells" or "smells") i.e., symptoms of poor design and implementation choices [27], represent one important factor contributing to technical debt, and possibly affecting the maintainability of a software system [42]. In the past and, most notably, in recent years, several studies investigated the relevance that code smells have for developers [68], [90], the extent to which code smells tend to remain in a software system for long periods of time [4], [17], [68], [66], as well as the side effects of code smells, such as an increase in change- and fault-proneness [37], [38] or decrease of software understandability [1] and maintainability [72], [88], [89]. While the repercussions of code smells on software quality have been empirically proven, there is still noticeable lack of empirical evidence related to how, when, and why they occur in software projects, as well as whether, after how long, and how they are removed [14]. This represents an obstacle for an effective and efficient management of technical debt. Also, understanding the typical lifecycle of code smells and the actions undertaken by developers to remove them is of paramount importance in the conception of recommender tools for developers' support. In other words, only a proper understanding of the phenomenon would allow the creation of recommenders able to highlight the presence of code smells and suggesting refactorings only when appropriate, hence avoiding information overload for developers [70].

Common wisdom suggests that urgent maintenance activities and pressure to deliver features while prioritizing time-to-market over code quality are often the causes of such smells. Generally speaking, software evolution has always been considered as one of the reasons behind "software aging" [61] or "increasing complexity" [44], [55], [87]. Also, one of the common beliefs is that developers remove code smells from the system by performing refactoring operations. However, to the best of our knowledge, there is no comprehensive empirical investigation into when and why code smells are introduced in software projects, how long they survive, and how they are removed.

In this paper we fill the void in terms of our understanding of code smells, reporting the results of a large-scale empirical study conducted on the change history of 200 open source projects belonging to three software ecosystems, namely Android, Apache and Eclipse. The study aims at investigating (i) when smells are introduced in software projects, (ii) why

**Code Smells have a high survivability and are rarely removed as a direct consequence of refactoring activities.**

**[Tufano et al. - IEEE TSE 2017]**

# So what ?

# More automation is needed !

**both to identify and refactor bad smells**

# Part I

## Bad Code Smells and Software Refactoring

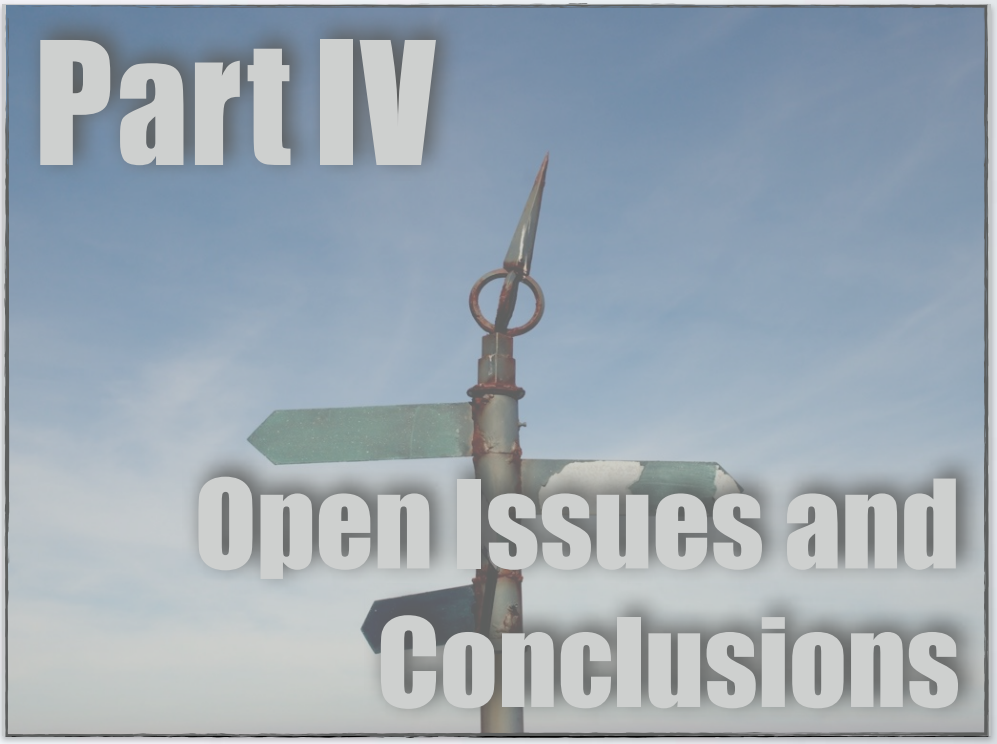
# Part II

## The refactoring process

# Part III

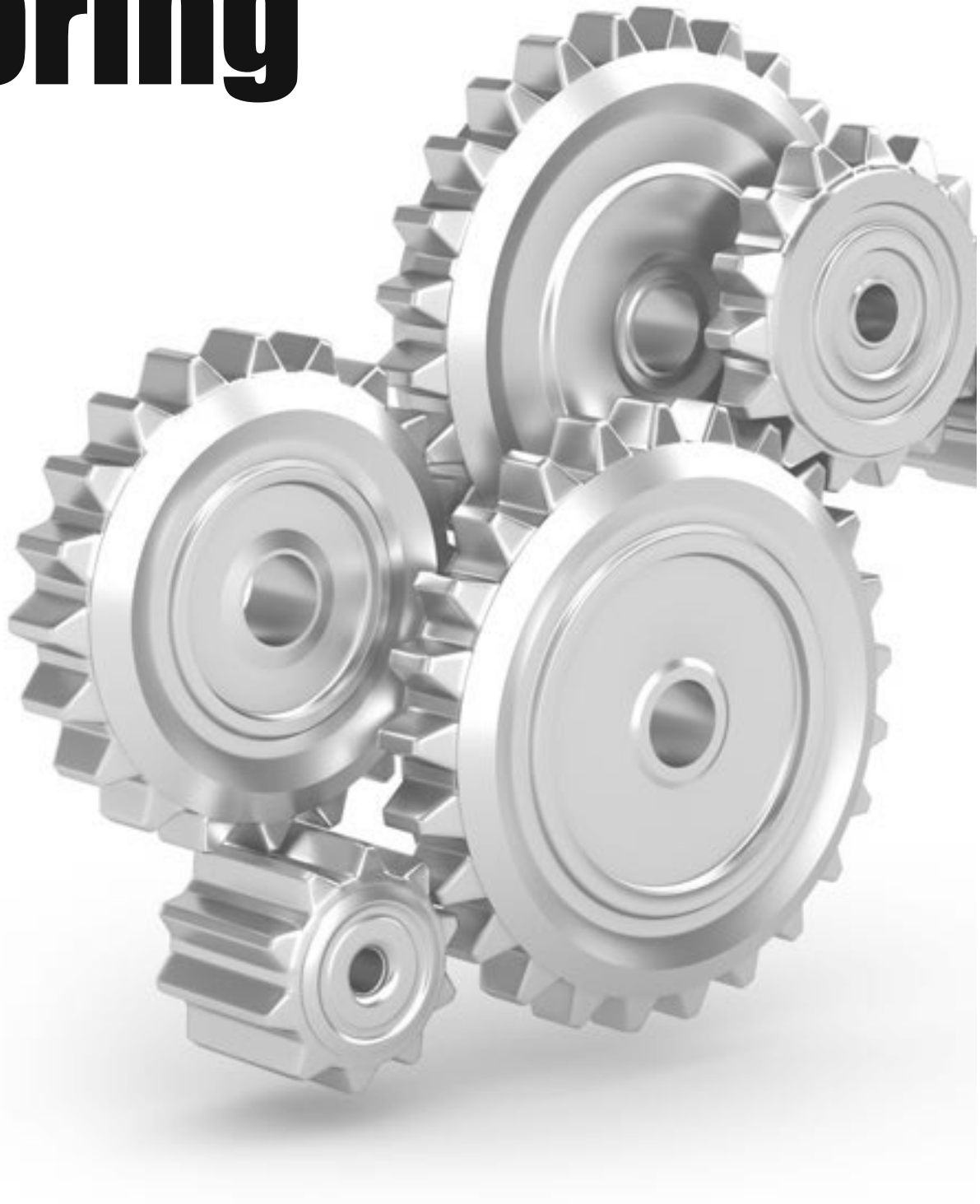
## Evaluating Refactoring Methods and Tools

# Part IV



## Open Issues and Conclusions

# The refactoring process





## A Survey of Software Refactoring

Tom Mens, *Member, IEEE*, and Tom Tourwé

**Abstract**—This paper provides an extensive overview of existing research in the field of software refactoring. This research is compared and discussed based on a number of different criteria: the refactoring activities that are supported, the specific techniques and formalisms that are used for supporting these activities, the types of software artifacts that are being refactored, the important issues that need to be taken into account when building refactoring tool support, and the effect of refactoring on the software process. A running example is used throughout the paper to explain and illustrate the main concepts.

**Index Terms**—Coding tools and techniques, programming environments/construction tools, restructuring, reverse engineering, and reengineering.

### 1 Introduction

An intrinsic property of software in a real-world environment is its need to evolve. As the software is enhanced, modified, and adapted to new requirements, the code becomes more complex and drifts away from its original design, thereby lowering the quality of the software. Because of this, the major part of the total software development cost is devoted to software maintenance [1], [2], [3]. Better software development methods and tools do not solve this problem because their increased capacity is used to implement more new requirements within the same time frame [4], making the software more complex again.

To cope with this spiral of complexity, there is an urgent need for techniques that reduce software complexity by incrementally improving the internal software quality. The research domain that addresses this problem is referred to as *restructuring* [5], [79] or, in the specific case of object-oriented software development, *refactoring* [6], [7].

According to the taxonomy of Chikofsky and Cross [8], *restructuring* is defined as "*the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics). A restructuring transformation is often one of appearance, such as altering code to improve its structure in the traditional sense of structured design. While restructuring creates new versions that implement or propose change to the subject system, it does not normally involve modifications because of new requirements. However, it may lead to better observations of the subject system that suggest changes that would improve aspects of the system.*"

The term *refactoring* was originally introduced by Opdyke in his PhD dissertation [6]. *Refactoring* is basically the object-oriented variant of restructuring: "*The process of changing a [object-oriented] software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure*" [7]. The key idea here is to redistribute classes, variables, and methods across the class hierarchy in order to facilitate future adaptations and extensions.

In the context of *software evolution*, restructuring and refactoring are used to improve the quality of the software (e.g., extensibility, modularity, reusability, complexity, maintainability, efficiency). Reledoring and restructuring are also used in the context of *reengineering* [9], which is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [8]. In this context, restructuring is needed to convert legacy code or deteriorated code into a more modular or structured form [10] or even to migrate code to a different programming language or even language paradigm [11].

The remainder of this paper is structured as follows: Section 2 explains general ideas of refactoring by means of an illustrative example. Section 3 identifies and explains the different refactoring activities. Section 4 provides an overview of various formalisms and techniques that can be used to support these refactoring activities. Section 5 summarizes different types of software artifacts for which refactoring support has been provided. Section 6 discusses essential issues that have to be considered in developing refactoring tools. Section 7 discusses how refactoring fits in the software development process. Finally, Section 8 concludes.

### 2 Running Example

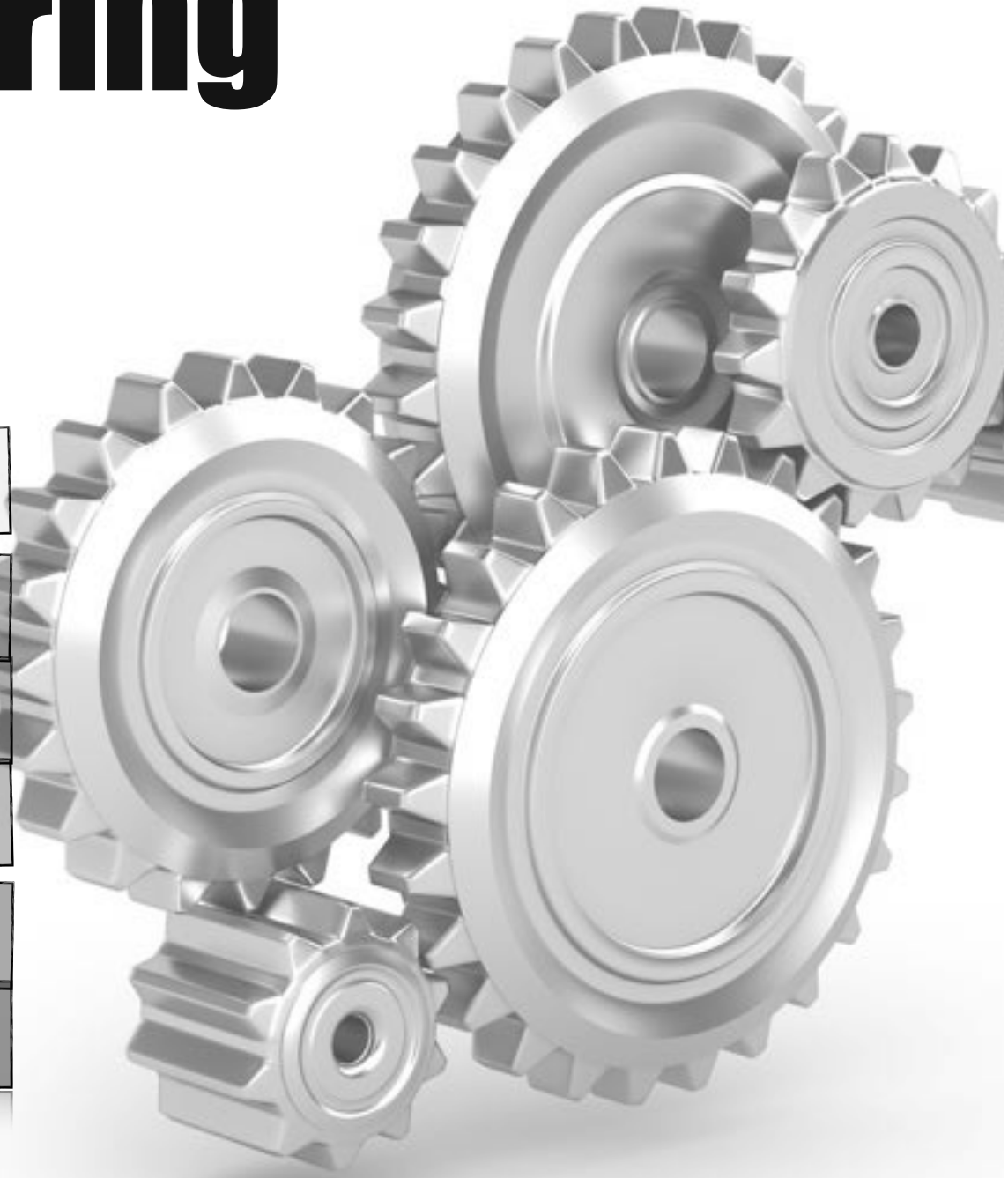
In this section, we introduce a running example that will be used throughout the paper. The example illustrates a typical nontrivial refactoring of an object-oriented design. The initial design depicted in Fig. 1 represents an object-oriented class hierarchy. It shows a *Document* class that is refined into three specific subclasses *ASCIIDoc*, *PSDoc*, and *PDFDoc*. A document provides *printing* and *print* facilities, which are realized by invoking the appropriate methods in the associated *Printer* and *Printer* classes, respectively. Before these methods can be invoked, some preprocessing or conversion needs to be done, which is realized differently for each of the *Document* subclasses. In Fig. 1, this is

• T. Mens is with the Université de Mons-Hainaut, Avenue du Champ de Mars 6, B-7000 Mons, Belgium. E-mail: tom.mens@umh.ac.be.
• T. Tourwé is with the Centrum voor Wiskunde en Informatica, PO Box 94079, NL-1090 GB Amsterdam, The Netherlands. E-mail: tom.tourwe@cwi.nl.

Manuscript received 30 Apr. 2003; revised 10 Dec. 2003; accepted 6 Jan. 2004. Recommended for acceptance by J.-M. Jézéquel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0047-0403.

0098-5589/04/$20.00 © 2004 IEEE Published by the IEEE Computer Society

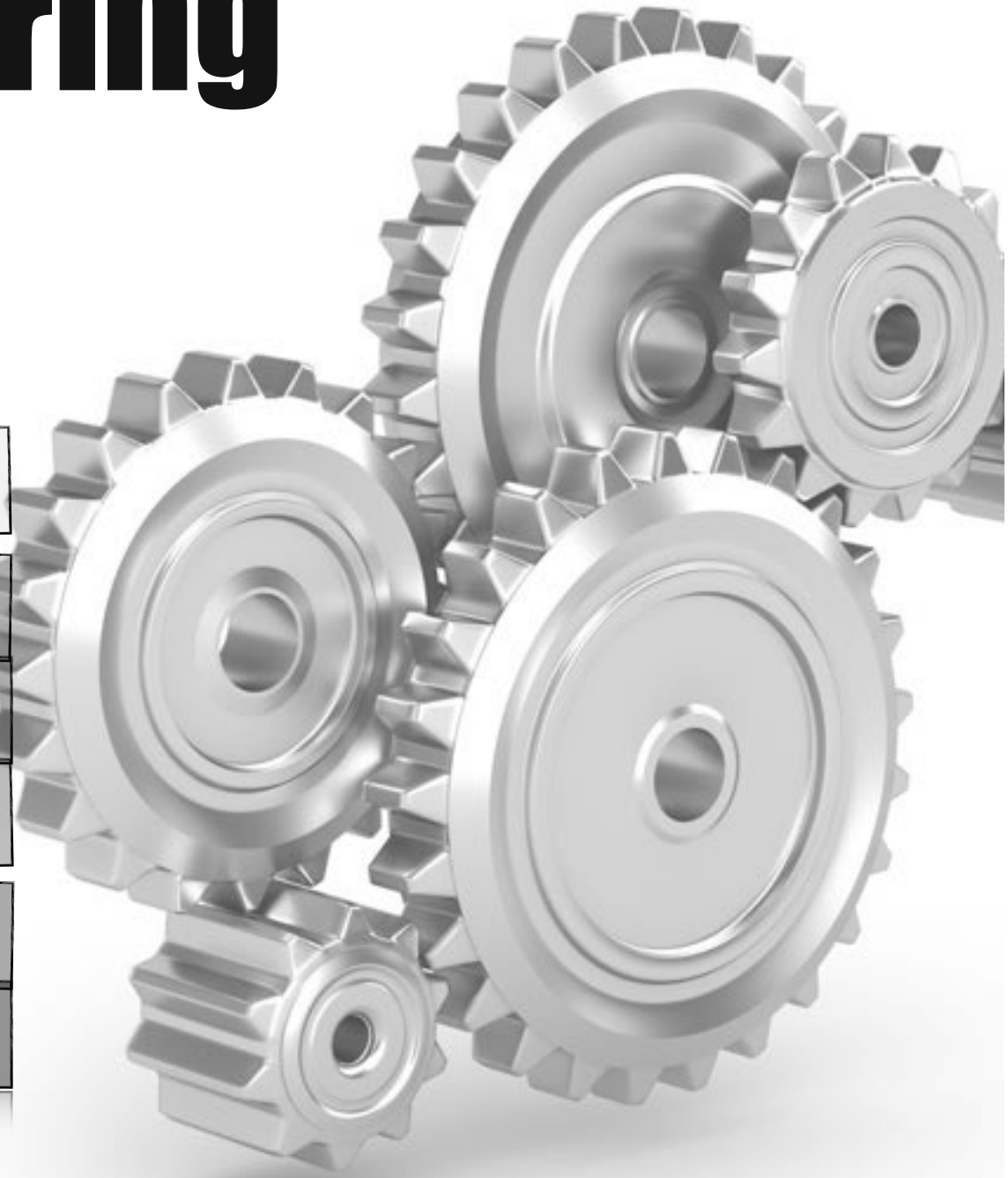
# The refactoring process



- Where to refactor
- How to refactor?
- Guarantee behaviour preservation
- Apply the refactoring
- Assess its effects on quality
- Consistently modify other artifacts

[Mens and Tourwé TSE 2004]

# The refactoring process



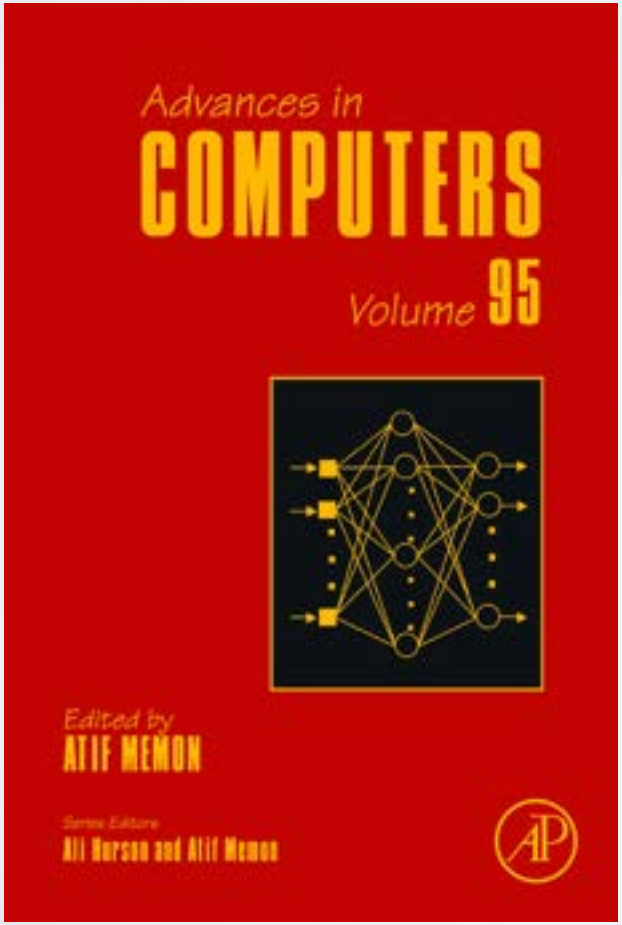
- Where to refactor
- How to refactor?
- Guarantee behaviour preservation
- Apply the refactoring
- Assess its effects on quality
- Consistently modify other artifacts

[Mens and Tourwé TSE 2004]

# Where to refactor

Google code smell detection

Scholar Circa 46.500 risultati (0,02 sec)




CHAPTER FOUR

## Anti-Pattern Detection: Methods, Challenges, and Open Issues

Fabio Palomba*, Andrea De Lucia*, Gabriele Bavota†, Rocco Oliveto‡

Contents

# Most approaches use structural information

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 35, NO. 3, MAY/JUNE 2009 347

## Identification of Move Method Refactoring Opportunities

Nikolaos Tsantalis, *Student Member, IEEE*, and Alexander Chatzigeorgiou, *Member, IEEE*

**Abstract**—Placement of attributes/methods within classes in an object-oriented system is usually guided by conceptual criteria and aided by appropriate metrics. Moving state and behavior between classes can help reduce coupling and increase cohesion, but it is nontrivial to identify where such refactorings should be applied. In this paper, we propose a methodology for the identification of Move Method refactoring opportunities that constitutes a way for solving many common Feature Envy bad smells. An algorithm that employs the notion of distance between system entities (attributes/methods) and classes extracts a list of behavior-preserving refactorings based on the examination of a set of preconditions. In practice, a software system may exhibit such problems in many different places. Therefore, our approach measures the effect of all refactoring suggestions based on a novel Entity Placement metric that quantifies how well entities have been placed in system classes. The proposed methodology can be regarded as a semi-automatic approach since the designer will eventually decide whether a suggested refactoring should be applied or not based on conceptual or other design quality criteria. The evaluation of the proposed approach has been performed considering qualitative, metric, conceptual, and efficiency aspects of the suggested refactorings in a number of open-source projects.

**Index Terms**—Move Method refactoring, Feature Envy, object-oriented design, Jaccard distance, design quality.

### 1 Introduction

According to several principles and laws of object-oriented design [18], [25], designers should always strive for low coupling and high cohesion. A number of empirical studies have investigated the relation of coupling and cohesion metrics with external quality indicators. Basili et al. [3] and Briand et al. [7] have shown that coupling metrics can serve as predictors of fault-prone classes. Briand et al. [8] and Chaumun et al. [12] have shown high positive correlation between the impact of changes (ripple effects, changeability) and coupling metrics. Brito e Abreu and Melo [11] have shown that Coupling Factor [10] has very high positive correlation with defect density and rework. Binkley and Schach [4] have shown that modules with low coupling (as measured by Coupling Dependency Metric) require less maintenance effort and have fewer maintenance faults and fewer runtime failures. Chidamber et al. [14] have shown that high levels of coupling and lack of cohesion are associated with lower productivity, greater rework, and greater design effort. Consequently, low coupling and high cohesion can be regarded as indicators of good design quality in terms of maintenance.

Coupling or cohesion problems manifest themselves in many different ways, with *Feature Envy* bad smell being the most common symptom. Feature Envy is a sign of violating the principle of grouping behavior with related data and occurs when a method is "more interested in a class other than the one it actually is in" [17]. Feature Envy problems can be solved in three ways [17]: 1) by moving a method to the class that it envies (Move Method refactoring); 2) by extracting a method fragment and then moving it to the class that it envies (Extract + Move Method refactoring); and 3) by moving an attribute to the class that envies it (Move Field refactoring). The correct application of the appropriate refactorings in a given system improves its design quality without altering its external behavior. However, the identification of methods, method fragments, or attributes that have to be moved to target classes is not always trivial since existing metrics may highlight coupling/cohesion problems but do not suggest specific refactoring opportunities.

Our methodology considers only Move Method refactorings as solutions to the Feature Envy design problem. Moving attributes (fields) from one class to another has not been considered, since this strategy would lead to contradicting refactoring suggestions with respect to the strategy of moving methods. Moreover, fields have stronger conceptual binding to the classes in which they are initially placed since they are less likely than methods to change once assigned to a class.

In this paper, the notion of distance between an entity (attribute or method) and a class is employed to support the automated identification of Feature Envy bad smells. To this end, an algorithm has been developed that extracts Move Method refactoring suggestions. For each method of the system, the algorithm forms a set of candidate target classes where the method can possibly be moved by examining the entities that it accesses from the system classes (system classes refer to the application or program under consideration excluding imported libraries or frameworks). Then, it iterates over the candidate target classes according to the number of accessed entities and the distance of the method from each candidate class. Eventually, it selects as the final

• The authors are with the Department of Applied Informatics, University of Macedonia, 54006 Thessaloniki, Greece.
E-mail: nikolaos@uom.gr, achat@uom.gr.

Manuscript received 15 Apr. 2008; revised 5 Dec. 2008; accepted 31 Dec. 2008; published online 6 Jan. 2009.
Recommended for acceptance by M. Chaudron.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2008-04-0150.
Digital Object Identifier no. 10.1109/TSE.2009.1.



Published by the IEEE Computer Society

20 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 36, NO. 1, JANUARY/FEBRUARY 2010

## 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.

• 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: mohaliou@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-BP. A, Park Plaza 59650 Villeneuve d'Ascq, France.
E-mail: {Laurence.Duchien, Anne-Francoise.Le_Meur@lifl.fr}.

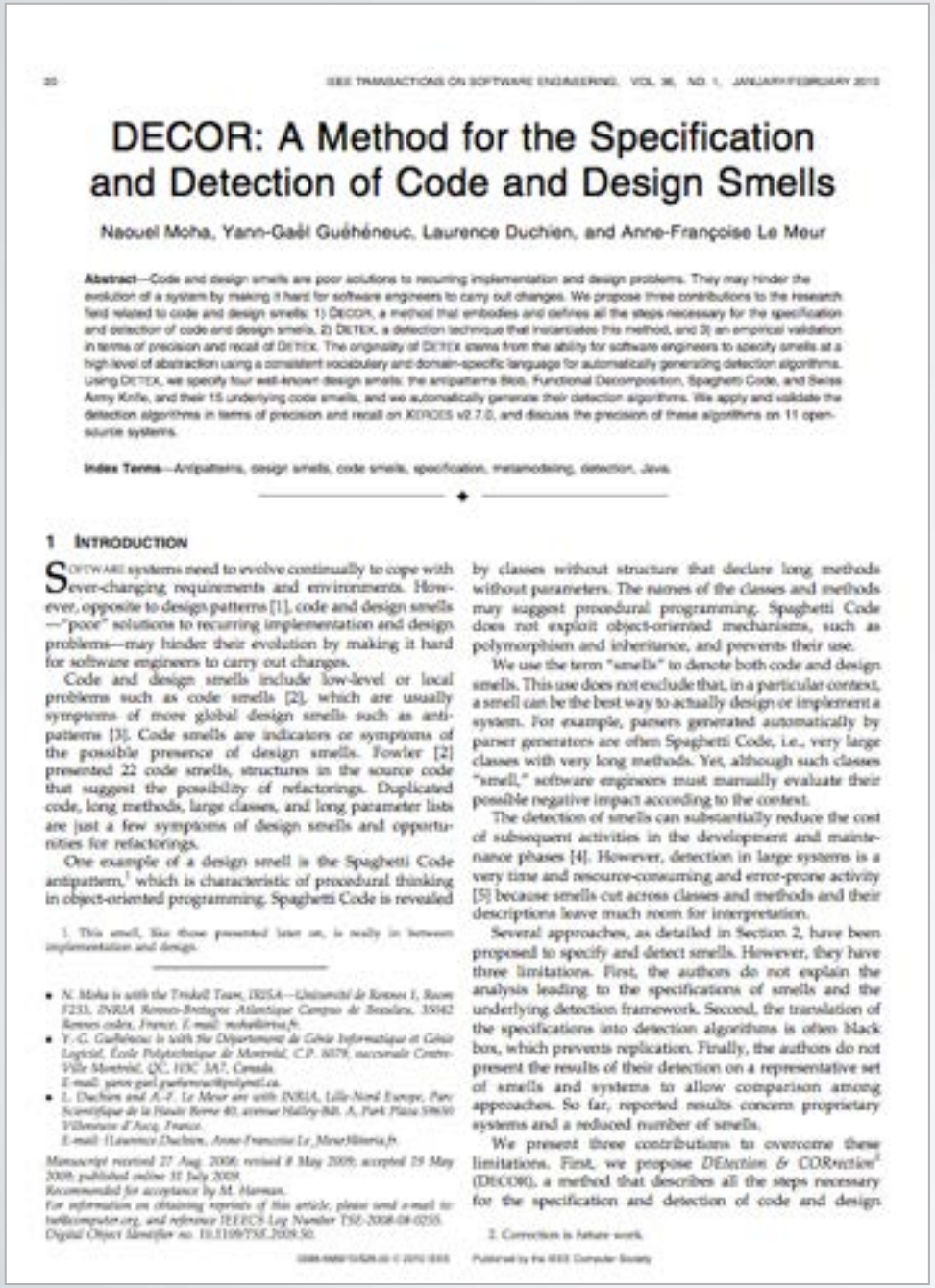
Manuscript received 27 Aug. 2008; revised 8 May 2009; accepted 19 May 2009; published online 31 July 2009.
Recommended for acceptance by M. Harman.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2008-08-0255.
Digital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



Published by the IEEE Computer Society

2. Correction is future work.

# DECOR



IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 36, NO. 1, JANUARY/FEBRUARY 2010

## 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

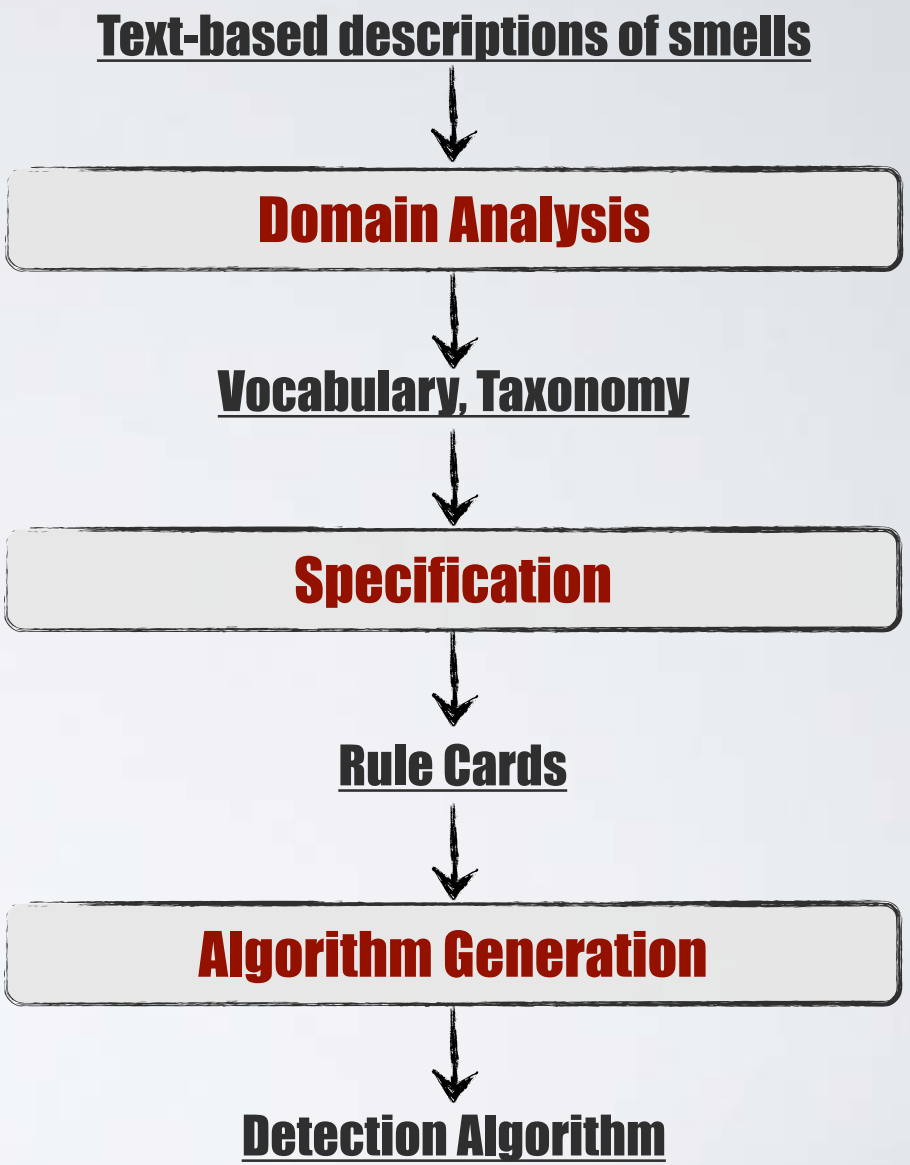
1. This smell, like those presented later on, is really in between implementation and design.


- 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: mohaboui@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-BP. 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.

2. Correction is future work.

Published by the IEEE Computer Society




[Moha et al. TSE 2010]

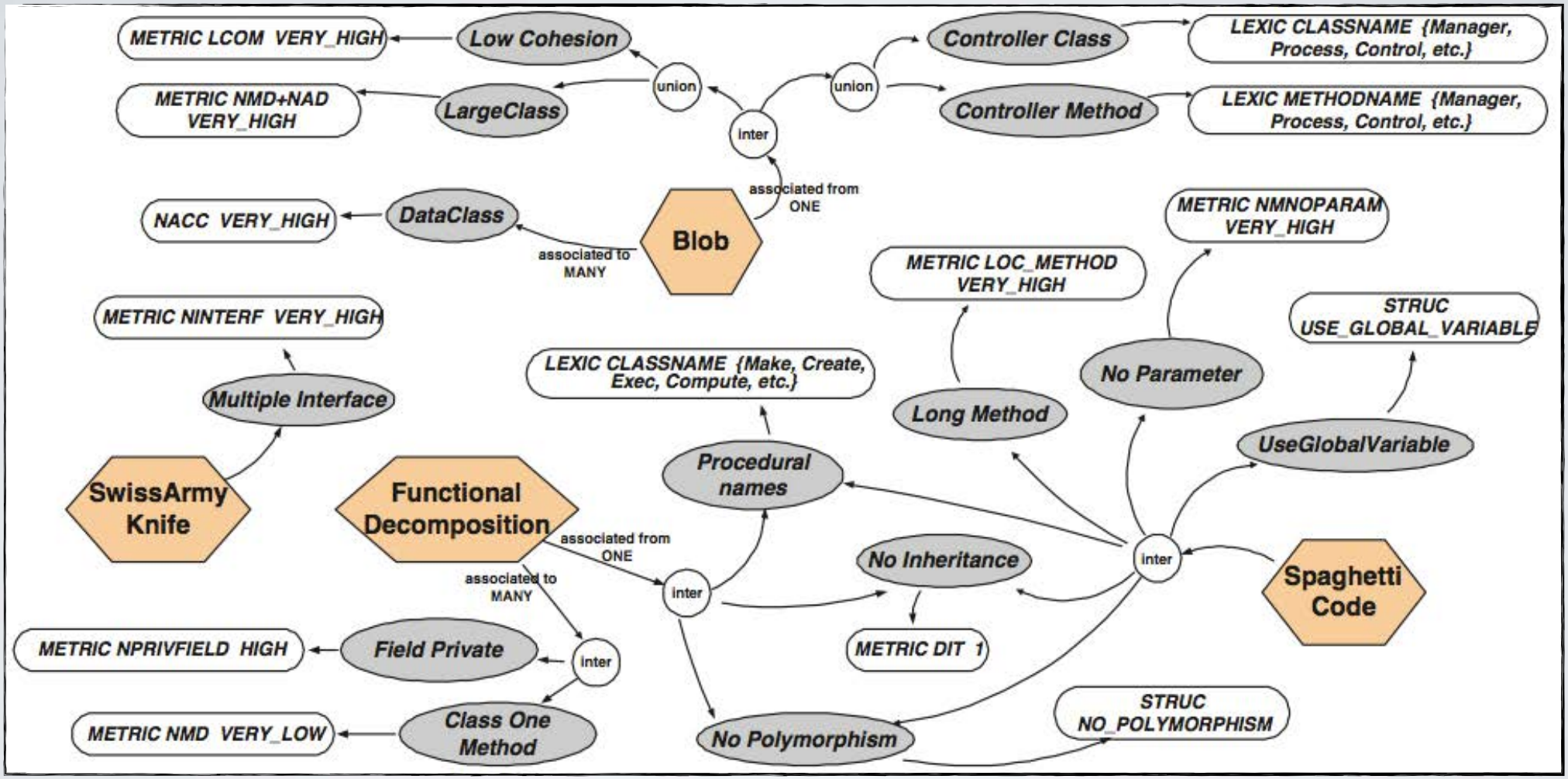
# DECOR

## 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) } ;

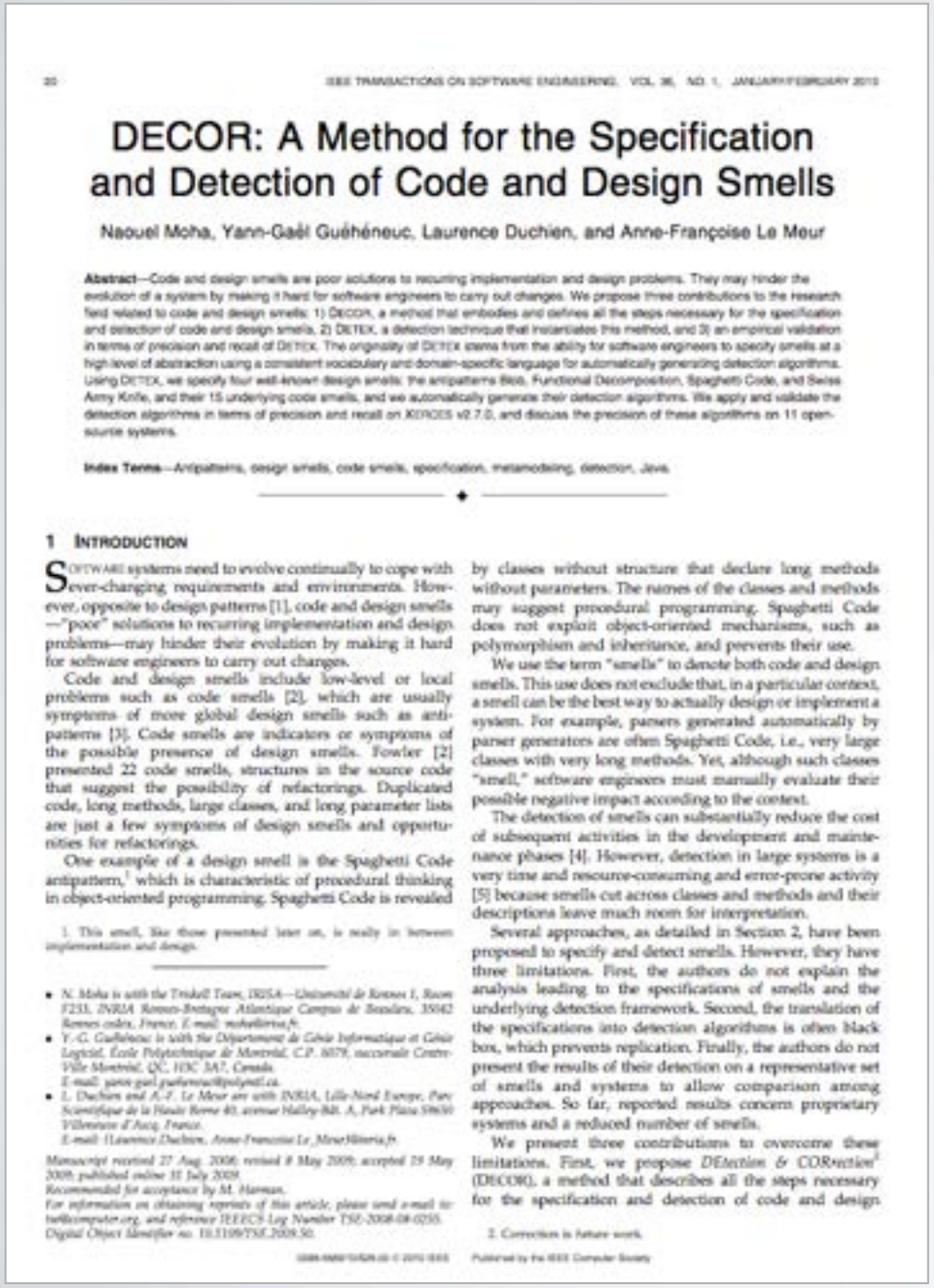
RULE : LowCohesion { ( METRIC : LCOM5,VERY_HIGH , 20) } ;

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



20 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 36, NO. 1, JANUARY/FEBRUARY 2010

## 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.


- 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: mohaboui@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-BP. 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.

2. Correction is future work.

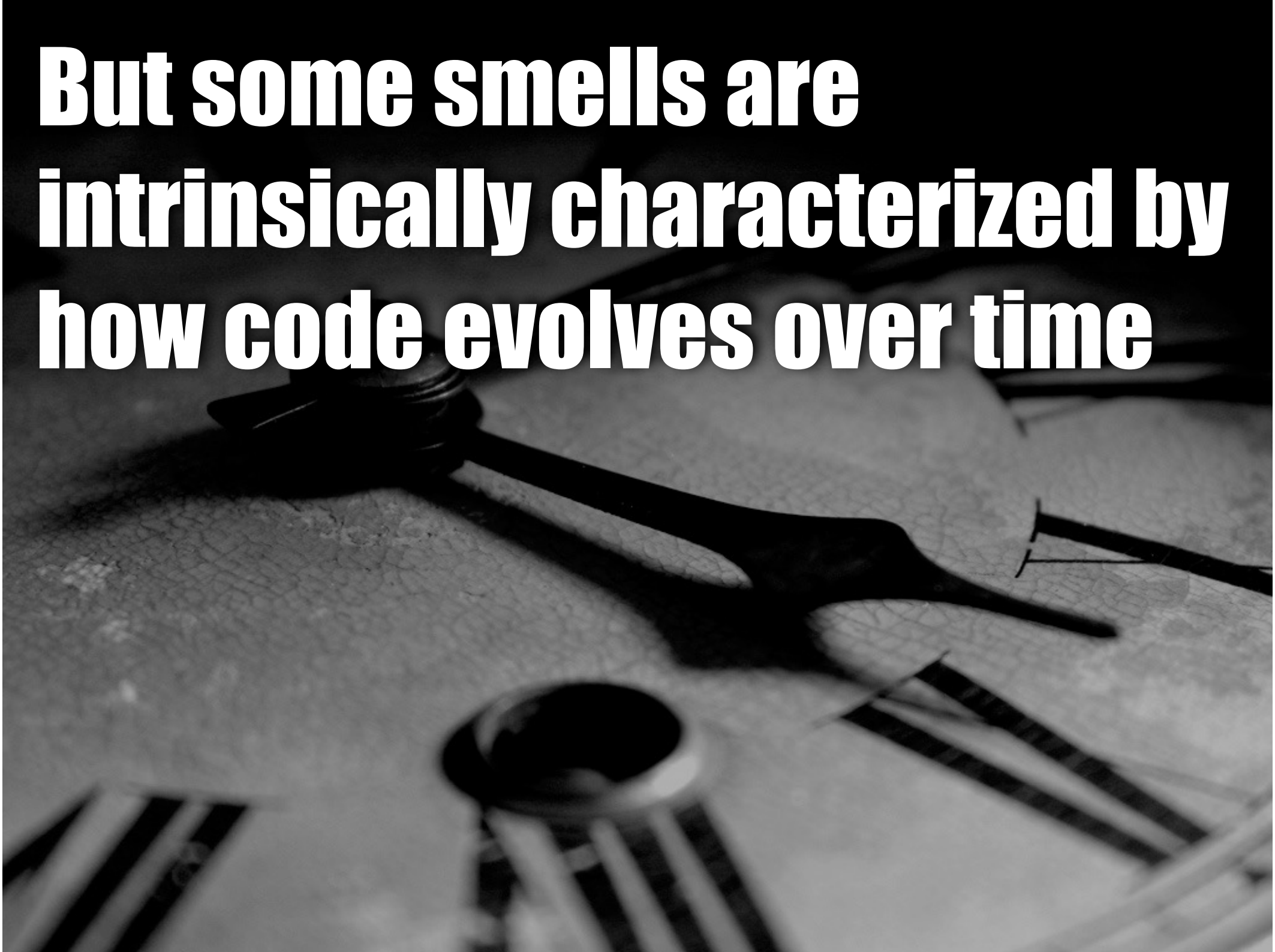
0098-5589/10/$26.00 © 2010 IEEE Published by the IEEE Computer Society

## 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]

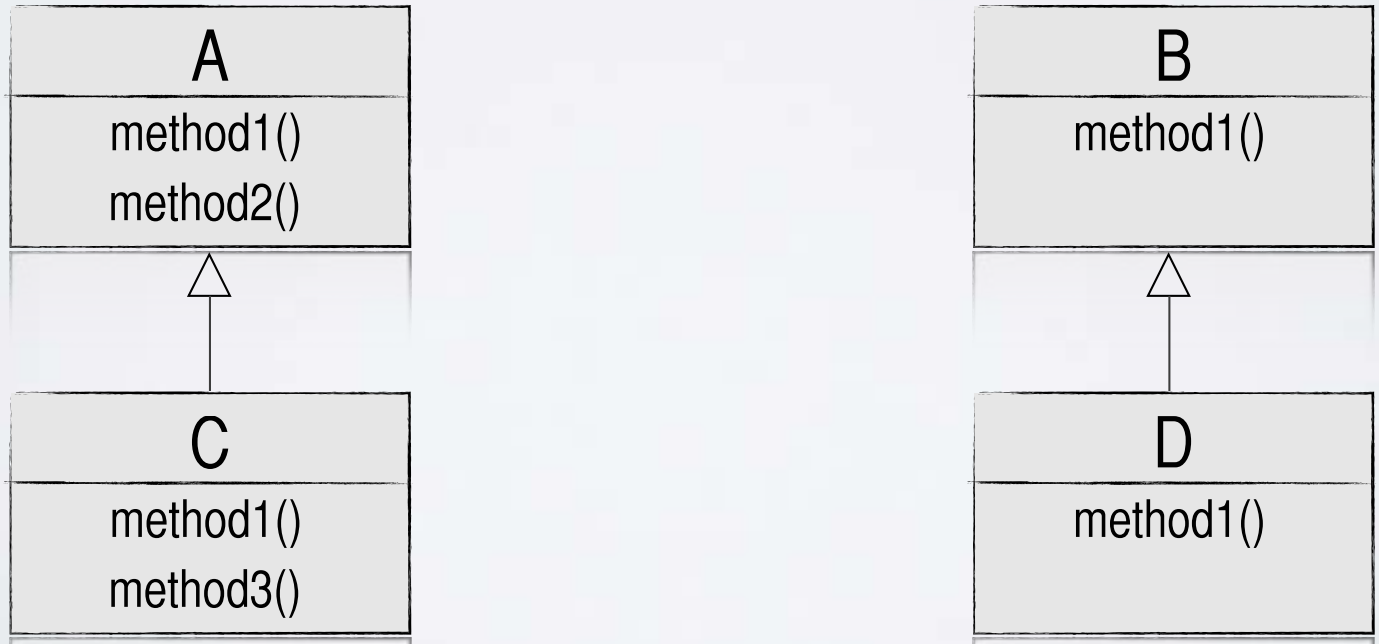

But some smells are intrinsically characterized by how code evolves over time

# Parallel Inheritance

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

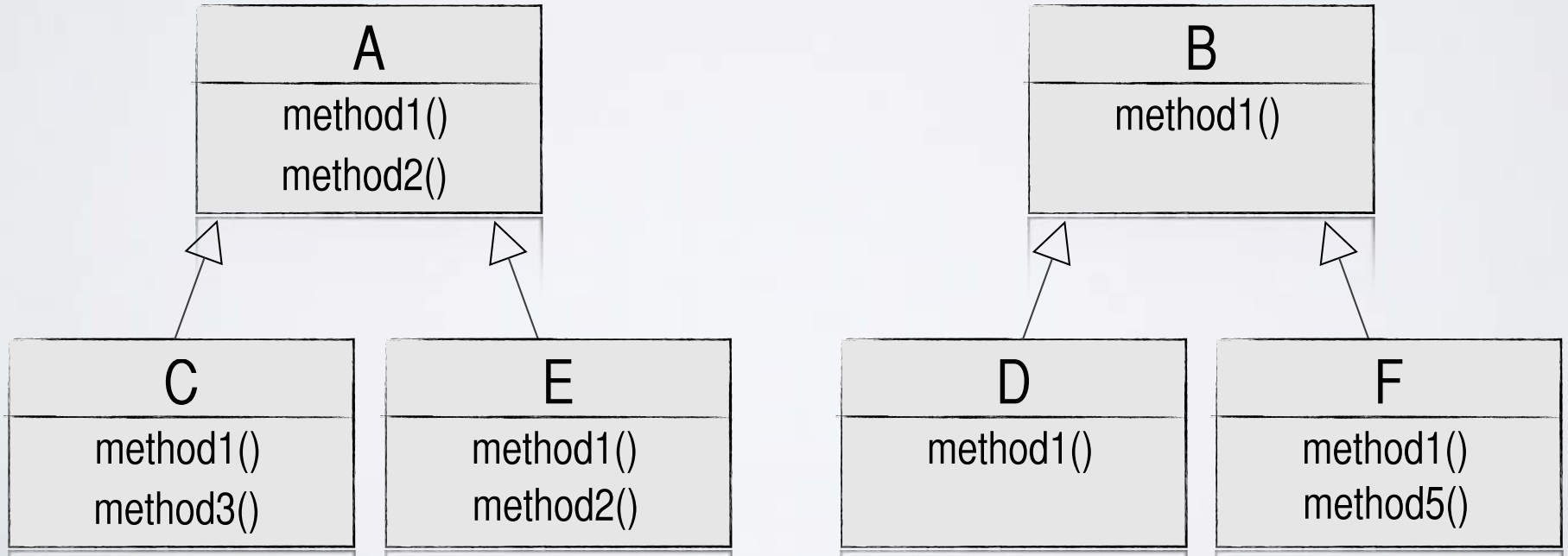
# Parallel Inheritance

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



# Historical Information for Smell deTection

# HIST

Mining Version Histories for Detecting Code Smells

Fabio Palomba, *Student Member, IEEE*, Gabriele Bavota, *Member, IEEE Computer Society*, Massimiliano Di Penta, Rocco Oliveto, *Member, IEEE Computer Society*, Denys Poshyvanyk, *Member, IEEE Computer Society*, and Andrea De Lucia, *Senior Member, IEEE*

**Abstract**—Code smells are symptoms of poor design and implementation choices that may hinder code comprehension, and possibly increase change- and fault-proneness. While most of the detection techniques just rely on structural information, many code smells are intrinsically characterized by how code elements change over time. In this paper, we propose Historical Information for Smell detection (HIST), an approach exploiting change history information to detect instances of five different code smells, namely Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy. We evaluate HIST in two empirical studies. The first, conducted on 20 open source projects, aimed at assessing the accuracy of HIST in detecting instances of the code smells mentioned above. The results indicate that the precision of HIST ranges between 72 and 86 percent, and its recall ranges between 58 and 100 percent. Also, results of the first study indicate that HIST is able to identify code smells that cannot be identified by competitive approaches solely based on code analysis of a single system's snapshot. Then, we conducted a second study aimed at investigating to what extent the code smells detected by HIST (and by competitive code analysis techniques) reflect developers' perception of poor design and implementation choices. We involved 12 developers of four open source projects that recognized more than 75 percent of the code smell instances identified by HIST as actual design/implementation problems.

**Index Terms**—Code smells, mining software repositories, empirical studies

## 1 Introduction

Code smells have been defined by Fowler [14] as symptoms of poor design and implementation choices. In some cases, such symptoms may originate from activities performed by developers while in a hurry, e.g., implementing urgent patches or simply making suboptimal choices. In other cases, smells come from some recurring, poor design solutions, also known as anti-patterns [10]. For example a *Blob* is a large and complex class that centralizes the behavior of a portion of a system and only uses other classes as data holders. *Blob* classes can rapidly grow out of control, making it harder and harder for developers to understand them, to fix bugs, and to add new features.

Previous studies have found that smells hinder comprehension [1], and possibly increase change- and fault-proneness [23], [24]. In summary, smells need to be carefully detected and monitored and, whenever necessary, refactoring actions should be planned and performed to deal with them.

There exist a number of approaches for detecting smells in source code to alert developers of their presence [30], [33], [48]. These approaches rely on structural information extracted from source code, for example, by means of constraints defined on some source code metrics. For instance, according to some existing approaches, such as *DECOR* [33], *LongMethod* or *LargeClass* smells are based on the size of the source code component in terms of LOC, whereas other smells like *ComplexClass* are based on the McCabe cyclomatic complexity [32]. Other smells, such as *Blob*, might use more complex rules.

Although existing approaches exhibit good detection accuracy, they still might not be adequate for detecting many of the smells described by Fowler [14]. In particular, there are some smells that, rather than being characterized by source code metrics or other information extracted from source code snapshots, are *intrinsically characterized by how source code changes over time*. For example, a *Parallel Inheritance* means that two or more class hierarchies evolve by adding code to both classes at the same time. Also, there are smells that are traditionally detected using structural information, where historical information can aid in capturing complementary, additionally useful properties. For example, a *Feature Envy* may manifest itself when a method of a class tends to change more frequently with methods of other classes rather than with those of the same class.

Based on such considerations, we propose an approach, named *Historical Information for Smell deTection* (HIST), to detect smells based on change history information mined from versioning systems, and, specifically, by analyzing co-changes occurring between source code artifacts. HIST is


- F. Palomba and A. De Lucia are with the University of Salerno, Fisciano (SA), Italy. E-mail: {fpalomba, adelucia}@unisa.it.
- G. Bavota is with the Free University of Bozen-Bolzano, Bolzano, Italy. E-mail: gabriele.bavota@unibz.it.
- M. Di Penta is with the University of Sannio, Benevento, Italy. E-mail: dipenta@unisannio.it.
- R. Oliveto is with the University of Molise, Pesche (IS), Italy. E-mail: rocco.oliveto@unimol.it.
- D. Poshyvanyk is with the College of William and Mary, Williamsburg, VA. E-mail: denys@cs.wm.edu.


*Manuscript received 21 May 2014; revised 24 Sept. 2014; accepted 16 Nov. 2014. Date of publication 19 Nov. 2014; date of current version 15 May 2015. Recommended for acceptance by A. Zeller.*
*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.2014.2372760*

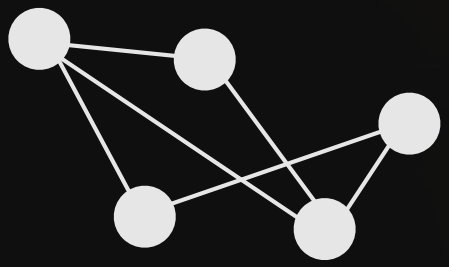


**[Palomba et al. - IEEE TSE 2015]**

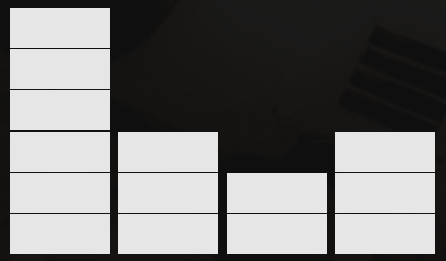
# HIST Change History Extractor

*log download* *code analyzer*

# HIST Code Smells Detector



*Association rule discovery to capture co-changes between entities*



*Analysis of change frequency of some specific entities*

# HIST Smells Detector



## Mining Version Histories for Detecting Code Smells

Fabio Palomba, *Student Member, IEEE*, Gabriele Bavota, *Member, IEEE Computer Society*, Massimiliano Di Penta, Rocco Oliveto, *Member, IEEE Computer Society*, Denys Poshyvanyk, *Member, IEEE Computer Society*, and Andrea De Lucia, *Senior Member, IEEE*

**Abstract**—Code smells are symptoms of poor design and implementation choices that may hinder code comprehension, and possibly increase change- and fault-proneness. While most of the detection techniques just rely on structural information, many code smells are intrinsically characterized by how code elements change over time. In this paper, we propose Historical Information for Smell detection (HIST), an approach exploiting change history information to detect instances of five different code smells, namely Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy. We evaluate HIST in two empirical studies. The first, conducted on 20 open source projects, aimed at assessing the accuracy of HIST in detecting instances of the code smells mentioned above. The results indicate that the precision of HIST ranges between 72 and 86 percent, and its recall ranges between 58 and 100 percent. Also, results of the first study indicate that HIST is able to identify code smells that cannot be identified by competitive approaches solely based on code analysis of a single system's snapshot. Then, we conducted a second study aimed at investigating to what extent the code smells detected by HIST (and by competitive code analysis techniques) reflect developers' perception of poor design and implementation choices. We involved 12 developers of four open source projects that recognized more than 75 percent of the code smell instances identified by HIST as actual design/implementation problems.

**Index Terms**—Code smells, mining software repositories, empirical studies.

### 1 Introduction

Code smells have been defined by Fowler [14] as symptoms of poor design and implementation choices. In some cases, such symptoms may originate from activities performed by developers while in a hurry, e.g., implementing urgent patches or simply making suboptimal choices. In other cases, smells come from some recurring, poor design solutions, also known as anti-patterns [10]. For example a *Blob* is a large and complex class that centralizes the behavior of a portion of a system and only uses other classes as data holders. *Blob* classes can rapidly grow out of control, making it harder and harder for developers to understand them, to fix bugs, and to add new features.

Previous studies have found that smells hinder comprehension [1], and possibly increase change- and fault-proneness [23], [24]. In summary, smells need to be carefully detected and monitored and, whenever necessary, refactoring actions should be planned and performed to deal with them.

There exist a number of approaches for detecting smells in source code to alert developers of their presence [30], [33], [40]. These approaches rely on structural information extracted from source code, for example, by means of constraints defined on some source code metrics. For instance, according to some existing approaches, such as *DECOR* [33], *LongMethod* or *LargeClass* smells are based on the size of the source code component in terms of LOC, whereas other smells like *ComplexClass* are based on the McCabe cyclomatic complexity [32]. Other smells, such as *Blob*, might use more complex rules.

Although existing approaches exhibit good detection accuracy, they still might not be adequate for detecting many of the smells described by Fowler [14]. In particular, there are some smells that, rather than being characterized by source code metrics or other information extracted from source code snapshots, are *intrinsically characterized by how source code changes over time*. For example, a *Parallel Inheritance* means that two or more class hierarchies evolve by adding code to both classes at the same time. Also, there are smells that are traditionally detected using structural information, where historical information can aid in capturing complementary, additionally useful properties. For example, a *Feature Envy* may manifest itself when a method of a class tends to change more frequently with methods of other classes rather than with those of the same class.

Based on such considerations, we propose an approach, named *Historical Information for Smell deTection* (HIST), to detect smells based on change history information mined from versioning systems, and, specifically, by analyzing co-changes occurring between source code artifacts. HIST is


- F. Palomba and A. De Lucia are with the University of Salerno, Fisciano (SA), Italy. E-mail: {fpalomba, adelucia}@unisa.it.
- G. Bavota is with the Free University of Bozen-Bolzano, Bolzano, Italy. E-mail: gabriele.bavota@unibz.it.
- M. Di Penta is with the University of Sannio, Benevento, Italy. E-mail: dipenta@unisannio.it.
- R. Oliveto is with the University of Molise, Pesche (IS), Italy. E-mail: rocco.oliveto@unimol.it.
- D. Poshyvanyk is with the College of William and Mary, Williamsburg, VA. E-mail: denys@cs.wm.edu.


*Manuscript received 21 May 2014; revised 24 Sept. 2014; accepted 16 Nov. 2014. Date of publication 19 Nov. 2014; date of current version 13 May 2015. Recommended for acceptance by A. Zeller.*
*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.2014.2372760*



**Detect instances of five code smells**

**Divergent Change,**
**Shotgun Surgery,**
**Parallel Inheritance ...**

**... but also**

**Blob**
**Feature Envy**

# Code Smells Detector

divergent change

**A class is changed in different ways for different reasons**

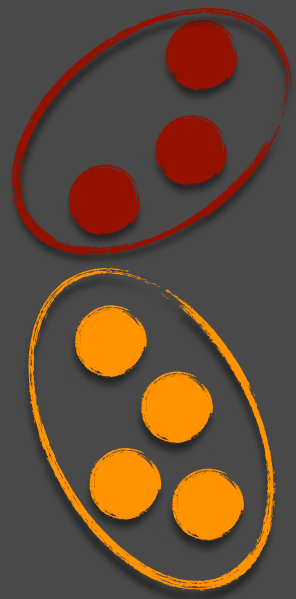
**Solution:**
**Extract Class Refactoring**

## Detection

**Classes containing at least two subsets 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

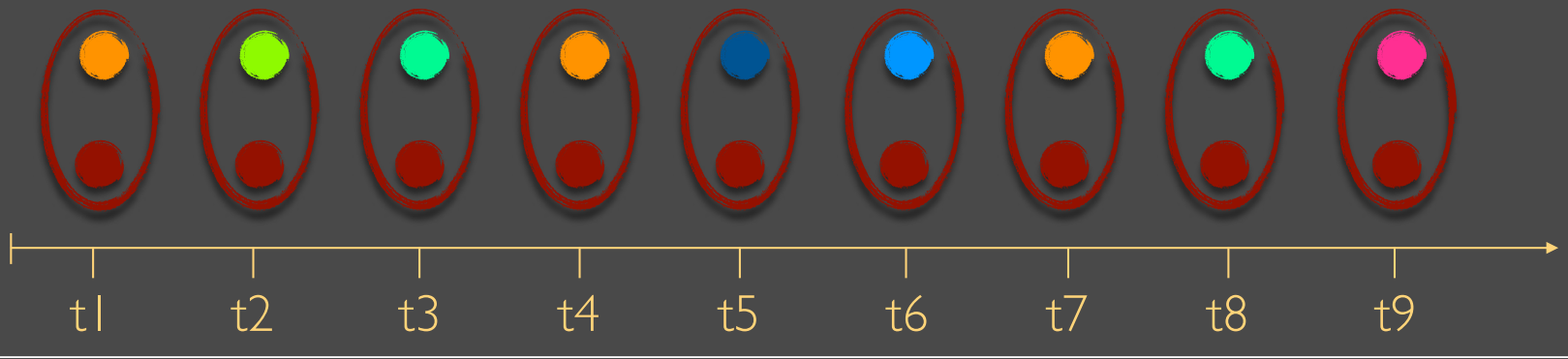
## blob

**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.**



# HIST: Evaluation



## Mining Version Histories for Detecting Code Smells

Fabio Palomba, *Student Member, IEEE*, Gabriele Bavota, *Member, IEEE Computer Society*, Massimiliano Di Penta, Rocco Oliveto, *Member, IEEE Computer Society*, Denys Poshyvanyk, *Member, IEEE Computer Society*, and Andrea De Lucia, *Senior Member, IEEE*

**Abstract**—Code smells are symptoms of poor design and implementation choices that may hinder code comprehension, and possibly increase change- and fault-proneness. While most of the detection techniques just rely on structural information, many code smells are intrinsically characterized by how code elements change over time. In this paper, we propose Historical Information for Smell detaTion (HIST), an approach exploiting change history information to detect instances of five different code smells, namely Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy. We evaluate HIST in two empirical studies. The first, conducted on 20 open source projects, aimed at assessing the accuracy of HIST in detecting instances of the code smells mentioned above. The results indicate that the precision of HIST ranges between 72 and 86 percent, and its recall ranges between 58 and 100 percent. Also, results of the first study indicate that HIST is able to identify code smells that cannot be identified by competitive approaches solely based on code analysis of a single system's snapshot. Then, we conducted a second study aimed at investigating to what extent the code smells detected by HIST (and by competitive code analysis techniques) reflect developers' perception of poor design and implementation choices. We involved 12 developers of four open source projects that recognized more than 75 percent of the code smell instances identified by HIST as actual design/implementation problems.

**Index Terms**—Code smells, mining software repositories, empirical studies.

### 1 Introduction

Code smells have been defined by Fowler [14] as symptoms of poor design and implementation choices. In some cases, such symptoms may originate from activities performed by developers while in a hurry, e.g., implementing urgent patches or simply making suboptimal choices. In other cases, smells arise from some recurring, poor design solutions, also known as anti-patterns [10]. For example a *Blob* is a large and complex class that centralizes the behavior of a portion of a system and only uses other classes as data holders. *Blob* classes can rapidly grow out of control, making it harder and harder for developers to understand them, to fix bugs, and to add new features.

Previous studies have found that smells hinder comprehension [1], and possibly increase change- and fault-proneness [23], [24]. In summary, smells need to be carefully detected and monitored and, whenever necessary, refactoring actions should be planned and performed to deal with them.

There exist a number of approaches for detecting smells in source code to alert developers of their presence [30], [33], [40]. These approaches rely on structural information extracted from source code, for example, by means of constraints defined on some source code metrics. For instance, according to some existing approaches, such as *DECOR* [33], *LongMethod* or *LargeClass* smells are based on the size of the source code component in terms of LOC, whereas other smells like *ComplexClass* are based on the McCabe cyclomatic complexity [32]. Other smells, such as *Blob*, might use more complex rules.

Although existing approaches exhibit good detection accuracy, they still might not be adequate for detecting many of the smells described by Fowler [14]. In particular, there are some smells that, rather than being characterized by source code metrics or other information extracted from source code snapshots, are *intrinsically characterized by how source code changes over time*. For example, a *Parallel Inheritance* means that two or more class hierarchies evolve by adding code to both classes at the same time. Also, there are smells that are traditionally detected using structural information, where historical information can aid in capturing complementary, additionally useful properties. For example, a *Feature Envy* may manifest itself when a method of a class tends to change more frequently with methods of other classes rather than with those of the same class.

Based on such considerations, we propose an approach, named *Historical Information for Smell deTection* (HIST), to detect smells based on change history information mined from versioning systems, and, specifically, by analyzing co-changes occurring between source code artifacts. HIST is


- *F. Palomba and A. De Lucia are with the University of Salerno, Fisciano (SA), Italy. E-mail: {fpalomba, adelucia}@unisa.it.*
- *G. Bavota is with the Free University of Bozen-Bolzano, Bolzano, Italy. E-mail: gabriele.bavota@unibz.it.*
- *M. Di Penta is with the University of Sannio, Benevento, Italy. E-mail: dipenta@unisannio.it.*
- *R. Oliveto is with the University of Molise, Pesche (IS), Italy. E-mail: rocco.oliveto@unimol.it.*
- *D. Poshyvanyk is with the College of William and Mary, Williamsburg, VA. E-mail: denys@cs.wm.edu.*


*Manuscript received 21 May 2014; revised 24 Sept. 2014; accepted 16 Nov. 2014. Date of publication 19 Nov. 2014; date of current version 13 May 2015. Recommended for acceptance by A. Zeller.*
*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.2014.2372760*



**Performances**

**Experimented on 20 systems and compared with different baseline tools (e.g., DECOR and JDeodorant)**

**Recall: 58-100%**
**Precision: 72-86%**

**Complementarity with baseline tools**

# HIST: Evaluation

462 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 41, NO. 5, MAY 2015

## Mining Version Histories for Detecting Code Smells

Fabio Palomba, *Student Member, IEEE*, Gabriele Bavota, *Member, IEEE Computer Society*, Massimiliano Di Penta, Rocco Oliveto, *Member, IEEE Computer Society*, Denys Poshyvanyk, *Member, IEEE Computer Society*, and Andrea De Lucia, *Senior Member, IEEE*

**Abstract**—Code smells are symptoms of poor design and implementation choices that may hinder code comprehension, and possibly increase change- and fault-proneness. While most of the detection techniques just rely on structural information, many code smells are intrinsically characterized by how code elements change over time. In this paper, we propose Historical Information for Smell detection (HIST), an approach exploiting change history information to detect instances of five different code smells, namely Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy. We evaluate HIST in two empirical studies. The first, conducted on 20 open source projects, aimed at assessing the accuracy of HIST in detecting instances of the code smells mentioned above. The results indicate that the precision of HIST ranges between 72 and 86 percent, and its recall ranges between 58 and 100 percent. Also, results of the first study indicate that HIST is able to identify code smells that cannot be identified by competitive approaches solely based on code analysis of a single system's snapshot. Then, we conducted a second study aimed at investigating to what extent the code smells detected by HIST (and by competitive code analysis techniques) reflect developers' perception of poor design and implementation choices. We involved 12 developers of four open source projects that recognized more than 75 percent of the code smell instances identified by HIST as actual design/implementation problems.

**Index Terms**—Code smells, mining software repositories, empirical studies.

### 1 Introduction

Code smells have been defined by Fowler [14] as symptoms of poor design and implementation choices. In some cases, such symptoms may originate from activities performed by developers while in a hurry, e.g., implementing urgent patches or simply making suboptimal choices. In other cases, smells arise from some recurring, poor design solutions, also known as anti-patterns [10]. For example a *Blob* is a large and complex class that centralizes the behavior of a portion of a system and only uses other classes as data holders. *Blob* classes can rapidly grow out of control, making it harder and harder for developers to understand them, to fix bugs, and to add new features.

Previous studies have found that smells hinder comprehension [1], and possibly increase change- and fault-proneness [23], [24]. In summary, smells need to be carefully detected and monitored and, whenever necessary, refactoring actions should be planned and performed to deal with them.

There exist a number of approaches for detecting smells in source code to alert developers of their presence [30], [33], [48]. These approaches rely on structural information extracted from source code, for example, by means of constraints defined on some source code metrics. For instance, according to some existing approaches, such as *DECOR* [33], *LongMethod* or *LargeClass* smells are based on the size of the source code component in terms of LOC, whereas other smells like *ComplexClass* are based on the McCabe cyclomatic complexity [32]. Other smells, such as *Blob*, might use more complex rules.

Although existing approaches exhibit good detection accuracy, they still might not be adequate for detecting many of the smells described by Fowler [14]. In particular, there are some smells that, rather than being characterized by source code metrics or other information extracted from source code snapshots, are *intrinsically characterized by how source code changes over time*. For example, a *Parallel Inheritance* means that two or more class hierarchies evolve by adding code to both classes at the same time. Also, there are smells that are traditionally detected using structural information, where historical information can aid in capturing complementary, additionally useful properties. For example, a *Feature Envy* may manifest itself when a method of a class tends to change more frequently with methods of other classes rather than with those of the same class.

Based on such considerations, we propose an approach, named *Historical Information for Smell deTection* (HIST), to detect smells based on change history information mined from versioning systems, and, specifically, by analyzing co-changes occurring between source code artifacts. HIST is


- F. Palomba and A. De Lucia are with the University of Salerno, Fisciano (SA), Italy. E-mail: {fpalomba, adelucia}@unisa.it.
- G. Bavota is with the Free University of Bozen-Bolzano, Bolzano, Italy. E-mail: gabriele.bavota@unibz.it.
- M. Di Penta is with the University of Sannio, Benevento, Italy. E-mail: dipenta@unisannio.it.
- R. Oliveto is with the University of Molise, Pesche (IS), Italy. E-mail: rocco.oliveto@unimol.it.
- D. Poshyvanyk is with the College of William and Mary, Williamsburg, VA. E-mail: denys@cs.wm.edu.


*Manuscript received 21 May 2014; revised 24 Sept. 2014; accepted 16 Nov. 2014. Date of publication 19 Nov. 2014; date of current version 13 May 2015. Recommended for acceptance by A. Zeller.*
*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.2014.2372760*



**User Study**

**12 developers of 4 open source systems where asked to evaluate the smells identified by HIST and baseline techniques**

**Over 75% of smells instances identified by HIST are considered as design/ implementation problems by developers**

# Code Smell Detection: New Frontiers

## The Use of Source Code Lexicon in the Context of Code Smell Detection

**A New Family of Software Anti-Patterns: Linguistic Anti-Patterns**

Venera Arnaoudova[1,2], Massimiliano Di Penta[3], Giuliano Antoniol[1], Yann-Gaël Guéhéneuc[1]
[1] *Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada*
[2] *Soccer Lab, DGIGL, École Polytechnique de Montréal, Canada*
[3] *Department of Engineering, University of Sannio, Benevento, Italy*
*E-mail: venera.arnaoudova@polymtl.ca, dipenta@unisannio.it, antoniol@ieee.org, yann-gael.gueheneuc@polymtl.ca*

**Abstract—Recent and past studies have shown that poor source code lexicon negatively affects software understandability, maintainability, and, overall, quality. Besides a poor usage of lexicon and documentation, sometimes a software artifact description is misleading with respect to its implementation. Consequently, developers will spend more time and effort when understanding these software artifacts, or even make wrong assumptions when they use them.**

**This paper introduces the definition of software linguistic antipatterns, and defines a family of them, i.e., those related to inconsistencies (i) between method signatures, documentation, and behavior and (ii) between attribute names, types, and comments. Whereas "design" antipatterns represent recurring, poor design choices, linguistic antipatterns represent recurring, poor naming and commenting choices.**

**The paper provides a first catalogue of one family of linguistic antipatterns, showing real examples of such antipatterns and explaining what kind of misunderstanding they can cause. Also, the paper proposes a detector prototype for Java programs called LAPD (Linguistic Anti-Pattern Detector), and reports a study investigating the presence of linguistic antipatterns in four Java software projects.**

***Keywords*-Software antipatterns, Source code lexicon, Textual analysis of software artifacts.**

### I. Introduction

Source code lexicon, i.e., the vocabulary used in naming software entities, is an essential element of any software system. A good source code lexicon can positively affect software quality, in particular comprehensibility and maintainability, and even reduce fault-proneness [1], [2], [3].

Several approaches have been developed for better lexicon and coding styles. Some researchers have developed approaches to assess the quality of source code lexicon [2], [4], [5], and some others provided a set of guidelines to produce high-quality identifiers [6].

In summary, existing literature analyzed the quality of source code lexicon solely in terms of what kinds of words were used, e.g., (i) whether identifiers are composed of words belonging to the English dictionary or to a domain specific dictionary; (ii) whether, instead, identifiers contain abbreviations, acronyms, and other combinations of characters. However, sometimes problems in the source code lexicon are more subtle and go beyond the occurrence of words. It may happen that the naming of a method does not properly reflect the method behavior, describing less (or more) than the method actually does. One such example, occurred in Eclipse 1.0, is a method named *isClassPathCorrect* defined in class *ProblemReporter*. One would expect that such a method returns a Boolean; instead, the method does not return any value and sets an attribute and calls another method to perform the task.

This paper represents the starting point for the definition of a new family of software antipatterns, named linguistic antipatterns. Software antipatterns—as they are known so far—are opposite to design patterns [7], i.e., they identify "poor" solutions to recurring design problems, for example, Brown's 40 antipatterns describe the most common pitfalls in the software industry [8]. They are generally introduced by developers not having sufficient knowledge and/or experience in solving a particular problem, or misusing good solutions, i.e., design patterns. Linguistic antipatterns shift the perspective from source code structure towards its consistency with the lexicon:

> **Linguistic Antipatterns (LAs)** in software systems are recurring poor practices in the naming, documentation, and choice of identifiers in the implementation of an entity, thus possibly impairing program understanding.

The presence of inconsistencies can be particularly harmful for developers that can make wrong assumptions about the code behavior or spend unnecessary time and effort to clarify it when understanding source code for their purposes. Therefore, highlighting their presence is essential for producing code easy to understand.

The contributions of this paper are:

1) A first catalogue of a family of LAs, focusing on inconsistencies between method/attribute naming conventions, documentation, and signature. For methods, such LAs are categorized into methods that (i) "do more than they say", (ii) "say more than they do", and (iii) "do the opposite than they say". Similarly, for attributes we categorize the LAs into attributes for which (i) "the name says more than the entity contains", (ii) "the name says less than the entity contains", and (iii) "the name says the opposite than the entity contains".
For each category, we report different LAs, explaining

"Linguistic Antipatterns are recurring poor practices in the naming, documentation, and choice of identifiers in the implementation of an entity,"

[Arnaudova et al. - CSMR 2013]

# TACO: Textual Analysis for Code smell detectiOn

A Textual-based Technique for Smell Detection

Fabio Palomba[1], Annibale Panichella[2], Andrea De Lucia[1], Rocco Oliveto[3], Andy Zaidman[2]
[1]University of Salerno, Italy – [2]Delft University of Technology, The Netherlands – [3]University of Molise, Italy

**Abstract—In this paper, we present TACO (Textual Analysis for Code Smell Detection), a technique that exploits textual analysis to detect a family of smells of different nature and different levels of granularity. We run TACO on 10 open source projects, comparing its performance with existing smell detectors purely based on structural information extracted from code components. The analysis of the results indicates that TACO's precision ranges between 67% and 77%, while its recall ranges between 72% and 84%. Also, TACO often outperforms alternative structural approaches confirming, once again, the usefulness of information that can be derived from the textual part of code components.**

I. Introduction

Continuous change requests, strict and close deadlines, the need to preserve the quality of source code to ease maintenance are just some of the challenges that developers must face every day. In such a scenario, finding the solution that provides the maximum gain from each point of view is quite impossible. Very often, due to time constraints or absence of software design documentation, developers decide to set aside good programming guidelines and implement a new change request in the most straightforward way. This way of working erodes the original design of the systems and introduces *technical debt* [1]. The erosion of the original design is generally represented by "*poor design or implementation choices*" [2], usually referred to as bad code smells (also named "code smells" or simply "smells"). Over the last decade, researchers investigated several aspects related to the presence of code smells, demonstrating (i) their relevance from the developers' perspective [3], [4], (ii) their longevity [5], [6], [7], [8], [9], and (iii) their impact on non-functional properties of source code, such as program comprehension [10], change- and fault-proneness [11], [12], and, more in general, on maintainability [13], [14], [15], [16]. For these reasons the research community devoted a lot of effort to define methods to detect code smells in source code and, whenever possible, trigger refactoring operations [17], [18], [19], [20], [21], [22], [23]. These tools generally apply constraint-based detection rules defined on some source code metrics, *i.e.*, the majority of existing approaches try to detect code smells through the analysis of structural properties of code components (e.g., methods).

Analyzing the catalogue of smells defined in the literature, it is easy to identify a specific family of smells that are represented by source code components with promiscuous responsibilities. For example, a *Blob* is a giant class that centralizes the behavior of a portion of the system and has a lot of different responsibilities, while a *Feature Envy* refers to a method more related to a different class with respect the one it is actually in. Even if these smells, such as Blob, are generally identified by considering structural properties of the code (see for instance [22]), there is still room for improving their detection by exploring other sources of information. Indeed, Palomba *et al.* [24] recently proposed the use of historical information for detecting several bad smells, including Blob. However, components with promiscuous responsibilities can be identified also considering the textual coherence of the source code vocabulary (i.e., terms extracted from comments and identifiers). Previous studies have indicated that lack of coherence in the code vocabulary can be successfully used to identify poorly cohesive [25] or more complex [26] classes. Following the same underlying assumption, in this paper we aim at investigating to what extent textual analysis can be used to detect smells related to promiscuous responsibilities. It is worth noting that textual analysis has already been used in several software engineering tasks [27], [28], [29], including refactoring [30], [31], [32]. However, our goal is to define an approach able to detect a family of code smells having different levels of granularity. To this aim, we define TACO (Textual Analysis for Code smell detectiOn), a smell detector purely based on Information Retrieval (IR) methods. We instantiated TACO for detecting five code smells, *i.e.*, *Long Method*, *Feature Envy*, *Blob*, *Promiscuous Package* and *Misplaced Class*. We conducted an empirical study involving 10 open source projects in order to (i) evaluate the accuracy of TACO when detecting code smells, and (ii) compare TACO with state-of-the-art structural-based detectors, namely DECOR [22], JDeodorant [23], and the approaches proposed in [33] and [34]. The results of our study indicate that TACO's precision ranges between 67% and 77%, while its recall is between 72% and 84%. When compared with the alternative structural-based detectors, we experienced that most of the times TACO outperforms these existing approaches. Finally, we observed some complementarities between textual and structural information suggesting that better performance can be achieved by combining the two sources of information.

II. Background and Related Work

Starting from the definition of design defects proposed in [2], [35], [36], [37], researchers have proposed semi-automated tools and techniques to detect code smells, such as ad-hoc manual inspection rules [38], tools to visualize code smells or refactoring opportunities [39], [40]. Further studies proposed to detect code smells by (i) identifying key *symptoms* that characterize particular bad smells using a set of thresholds based on the measurement of structural metrics (e.g., if Lines Of Code $> k$); (ii) conflating the identified symptoms, leading to the final rule for detecting the

978-1-5090-1428-6/16/$31.00 ©2016 IEEE

ICPC 2016, Austin, Texas

"Conceptual properties can provide complementary information to structural properties when identifying code smells in source code."

[Palomba et. al. - ICPC 2016]

# Textual 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);
 }
```

# Textual Information

```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);
 }
```

# Analysis of Textual Similarity between Code Components: Conjectures

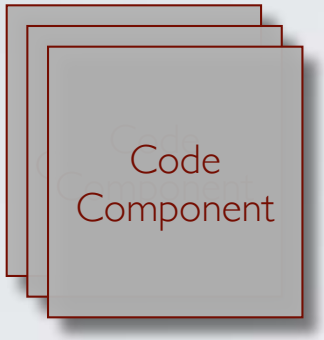
**If developers use similar terms in comments and identifiers of two code components, it is likely they implement similar responsibilities**

**Code components affected by smells contains unrelated textual content**

Conceptual Cohesion and Coupling Metrics defined by Denys Poshyvanyk et al.

# TACO Process

## Text Preprocessing



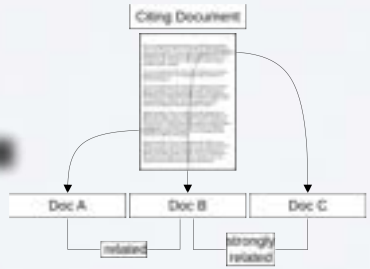

textual component extractor

Stemming
Term separation
…
Stop word removal

IR normalization process

## Smell Detector

block extractor

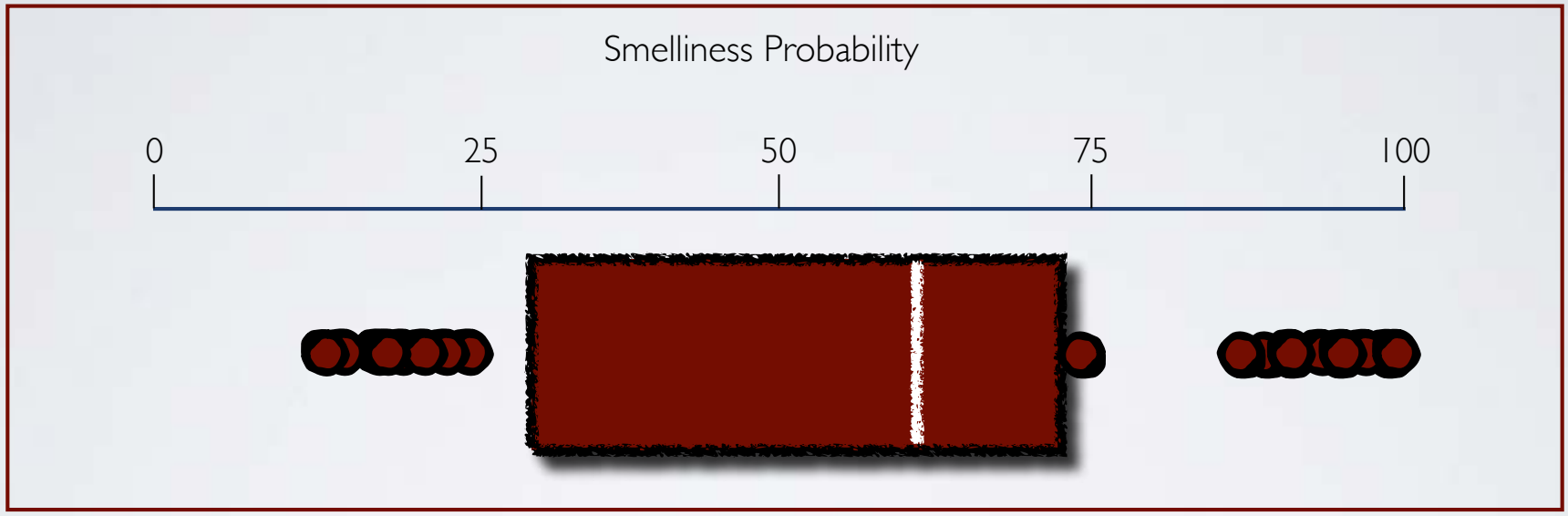



dissimilarity computation

**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

We instantiate TACO to detect 5 different code smells characterized by promiscuous responsibilities

Long Method

Blob

Promiscuous Package

# We instantiate TACO to 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

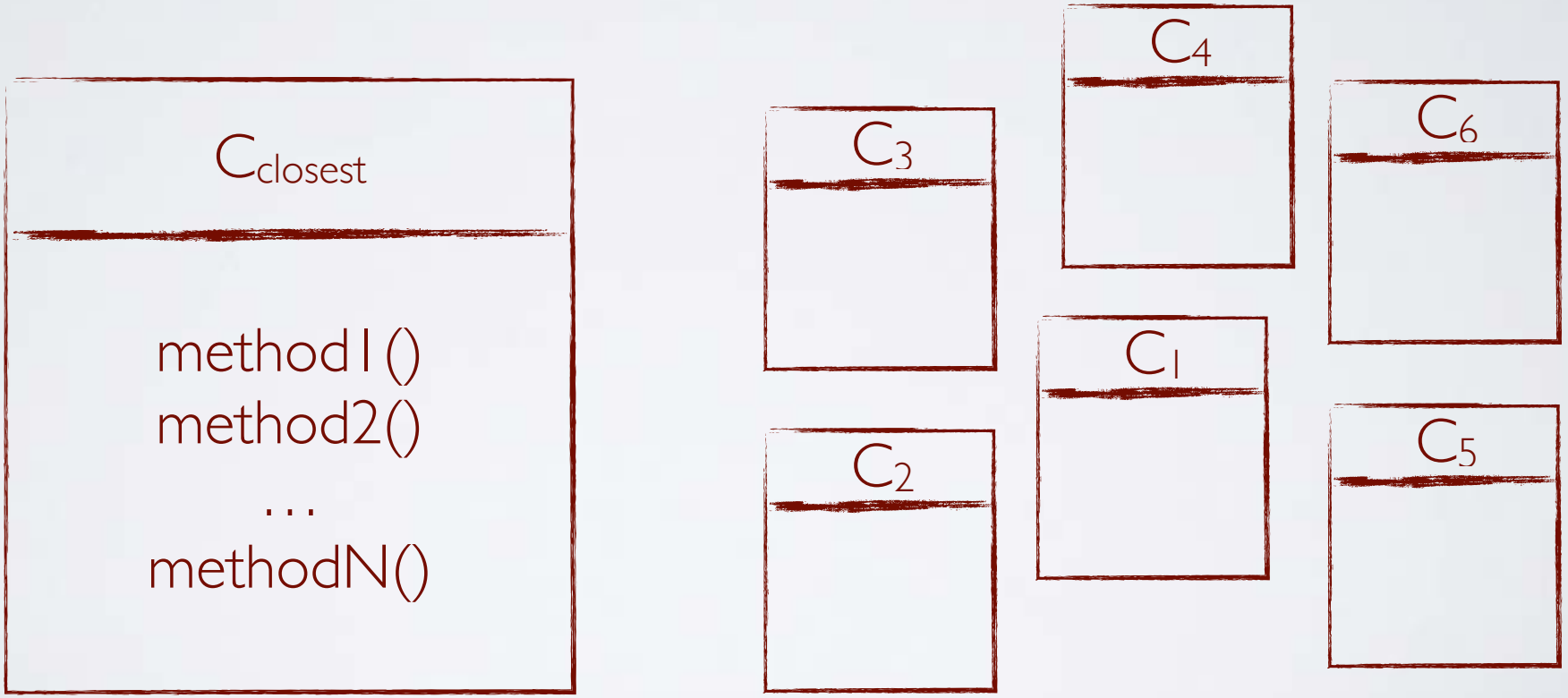
Long Method Probability Computation

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

# Detecting Feature Envy instances

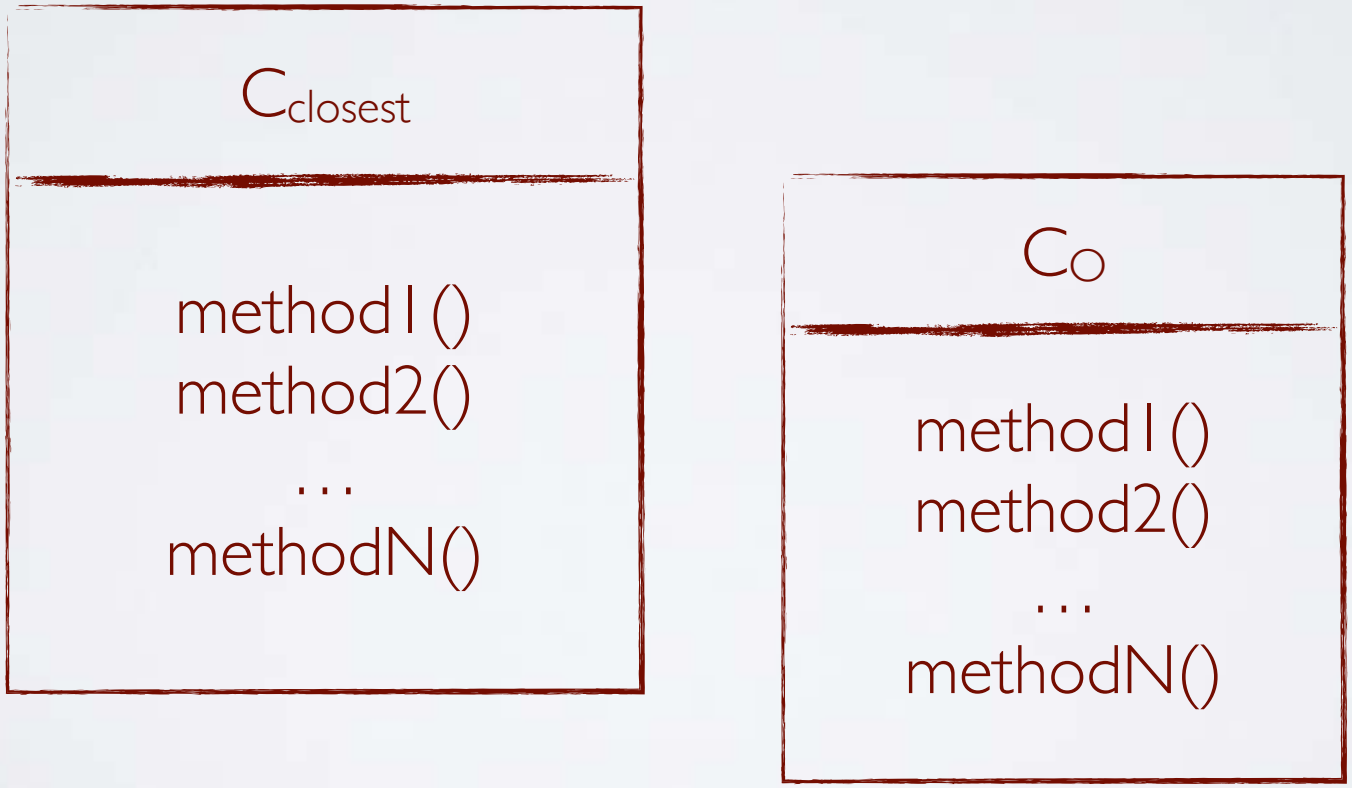
# Detecting Feature Envy instances

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



# Detecting Feature Envy instances

Feature Envy Probability Computation



# TACO: Evaluation

A Textual-based Technique for Smell Detection

Fabio Palomba[1], Annibale Panichella[2], Andrea De Lucia[1], Rocco Oliveto[3], Andy Zaidman[2]
[1]University of Salerno, Italy – [2]Delft University of Technology, The Netherlands – [3]University of Molise, Italy

**Abstract—In this paper, we present TACO (Textual Analysis for Code Smell Detection), a technique that exploits textual analysis to detect a family of smells of different nature and different levels of granularity. We run TACO on 10 open source projects, comparing its performance with existing smell detectors purely based on structural information extracted from code components. The analysis of the results indicates that TACO's precision ranges between 67% and 77%, while its recall ranges between 72% and 84%. Also, TACO often outperforms alternative structural approaches confirming, once again, the usefulness of information that can be derived from the textual part of code components.**

I. Introduction

Continuous change requests, strict and close deadlines, the need to preserve the quality of source code to ease maintenance are just some of the challenges that developers must face every day. In such a scenario, finding the solution that provides the maximum gain from each point of view is quite impossible. Very often, due to time constraints or absence of software design documentation, developers decide to set aside good programming guidelines and implement a new change request in the most straightforward way. This way of working erodes the original design of the systems and introduces *technical debt* [1]. The erosion of the original design is generally represented by *"poor design or implementation choices"* [2], usually referred to as bad code smells (also named "code smells" or simply "smells"). Over the last decade, researchers investigated several aspects related to the presence of code smells, demonstrating (i) their relevance from the developers' perspective [3], [4], (ii) their longevity [5], [6], [7], [8], [9], and (iii) their impact on non-functional properties of source code, such as program comprehension [10], change- and fault-proneness [11], [12], and, more in general, on maintainability [13], [14], [15], [16]. For these reasons the research community devoted a lot of effort to define methods to detect code smells in source code and, whenever possible, trigger refactoring operations [17], [18], [19], [20], [21], [22], [23]. These tools generally apply constraint-based detection rules defined on some source code metrics, *i.e.*, the majority of existing approaches try to detect code smells through the analysis of structural properties of code components (e.g., methods).

Analyzing the catalogue of smells defined in the literature, it is easy to identify a specific family of smells that are represented by source code components with promiscuous responsibilities. For example, a *Blob* is a giant class that centralizes the behavior of a portion of the system and has a lot of different responsibilities, while a *Feature Envy* refers to a method more related to a different class with respect the one it is actually in. Even if these smells, such as Blob, are generally identified by considering structural properties of the code (see for instance [22]), there is still room for improving their detection by exploring other sources of information. Indeed, Palomba *et al.* [24] recently proposed the use of historical information for detecting several bad smells, including Blob. However, components with promiscuous responsibilities can be identified also considering the textual coherence of the source code vocabulary (*i.e.*, terms extracted from comments and identifiers). Previous studies have indicated that lack of coherence in the code vocabulary can be successfully used to identify poorly cohesive [25] or more complex [26] classes. Following the same underlying assumption, in this paper we aim at investigating to what extent textual analysis can be used to detect smells related to promiscuous responsibilities. It is worth noting that textual analysis has already been used in several software engineering tasks [27], [28], [29], including refactoring [30], [31], [32]. However, our goal is to define an approach able to detect a family of code smells having different levels of granularity. To this aim, we define TACO (Textual Analysis for Code smell detectiOn), a smell detector purely based on Information Retrieval (IR) methods. We instantiated TACO for detecting five code smells, *i.e.*, *Long Method*, *Feature Envy*, *Blob*, *Promiscuous Package* and *Misplaced Class*. We conducted an empirical study involving 10 open source projects in order to (i) evaluate the accuracy of TACO when detecting code smells, and (ii) compare TACO with state-of-the-art structural-based detectors, namely DECOR [22], JDeodorant [23], and the approaches proposed in [33] and [34]. The results of our study indicate that TACO's precision ranges between 67% and 77%, while its recall is between 72% and 84%. When compared with the alternative structural-based detectors, we experienced that most of the times TACO outperforms these existing approaches. Finally, we observed some complementarities between textual and structural information suggesting that better performance can be achieved by combining the two sources of information.

II. Background and Related Work

Starting from the definition of design defects proposed in [2], [35], [36], [37], researchers have proposed semi-automated tools and techniques to detect code smells, such as ad-hoc manual inspection rules [38], tools to visualize code smells or refactoring opportunities [39], [40]. Further studies proposed to detect code smells by (i) identifying key *symptoms* that characterize particular bad smells using a set of thresholds based on the measurement of structural metrics (e.g., if Lines Of Code $\geq k$); (ii) conflating the identified symptoms, leading to the final rule for detecting the

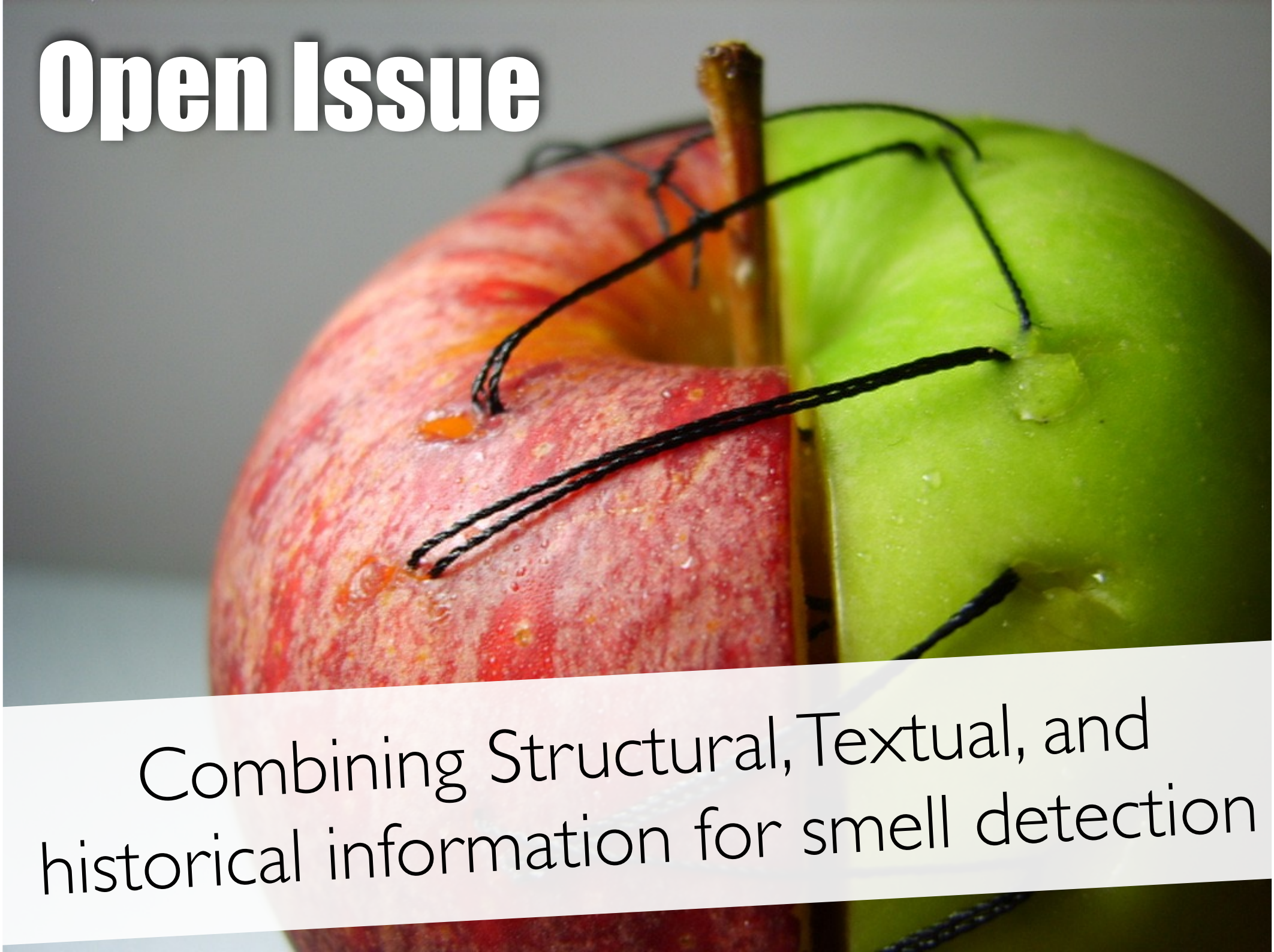
**Performances**

**Experimented on 10 systems and compared with different baseline tools (e.g., DECOR and JDeodorant)**

**+22% F-measure on average**

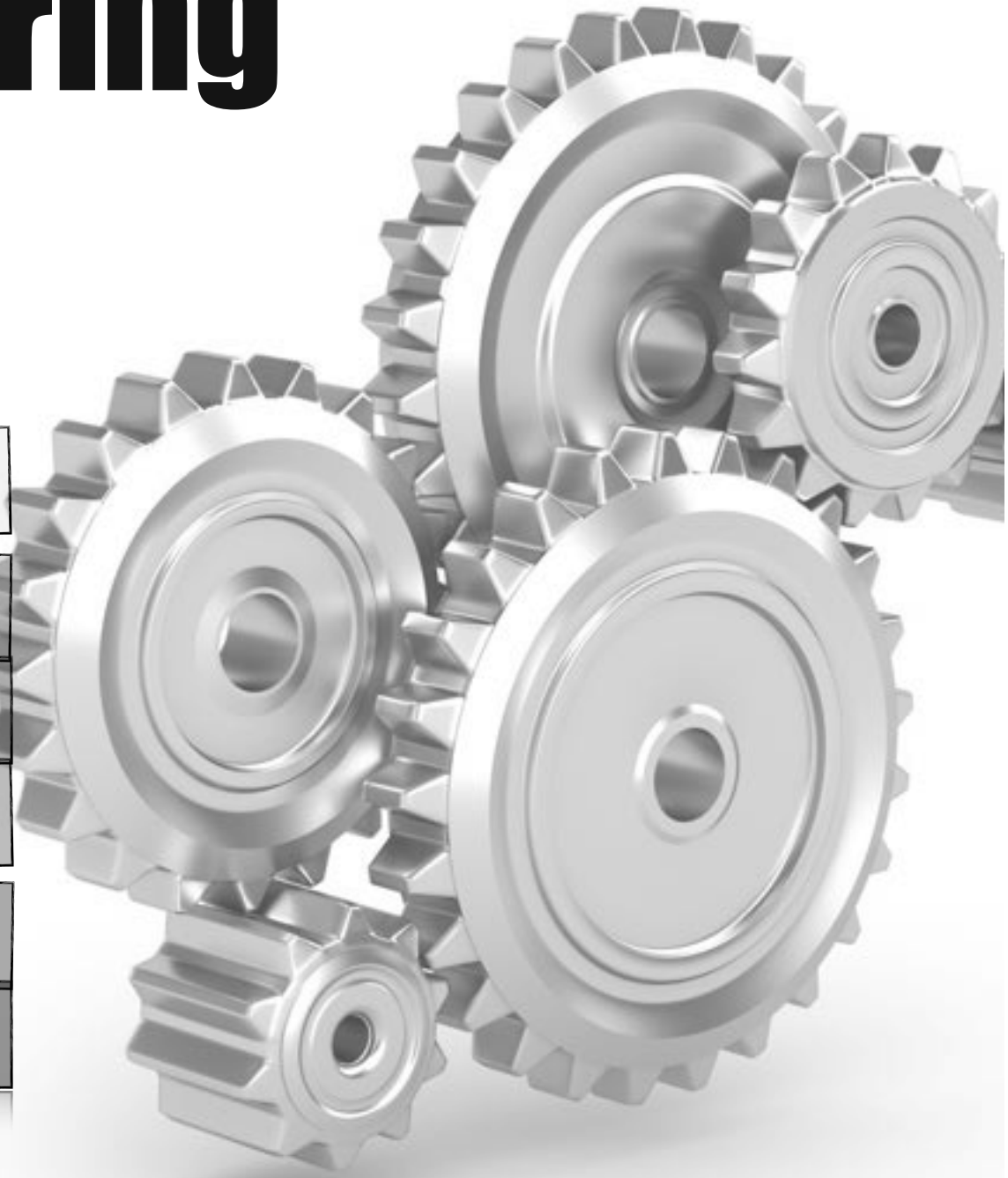
**Complementarity with baseline tools**

# Open Issue



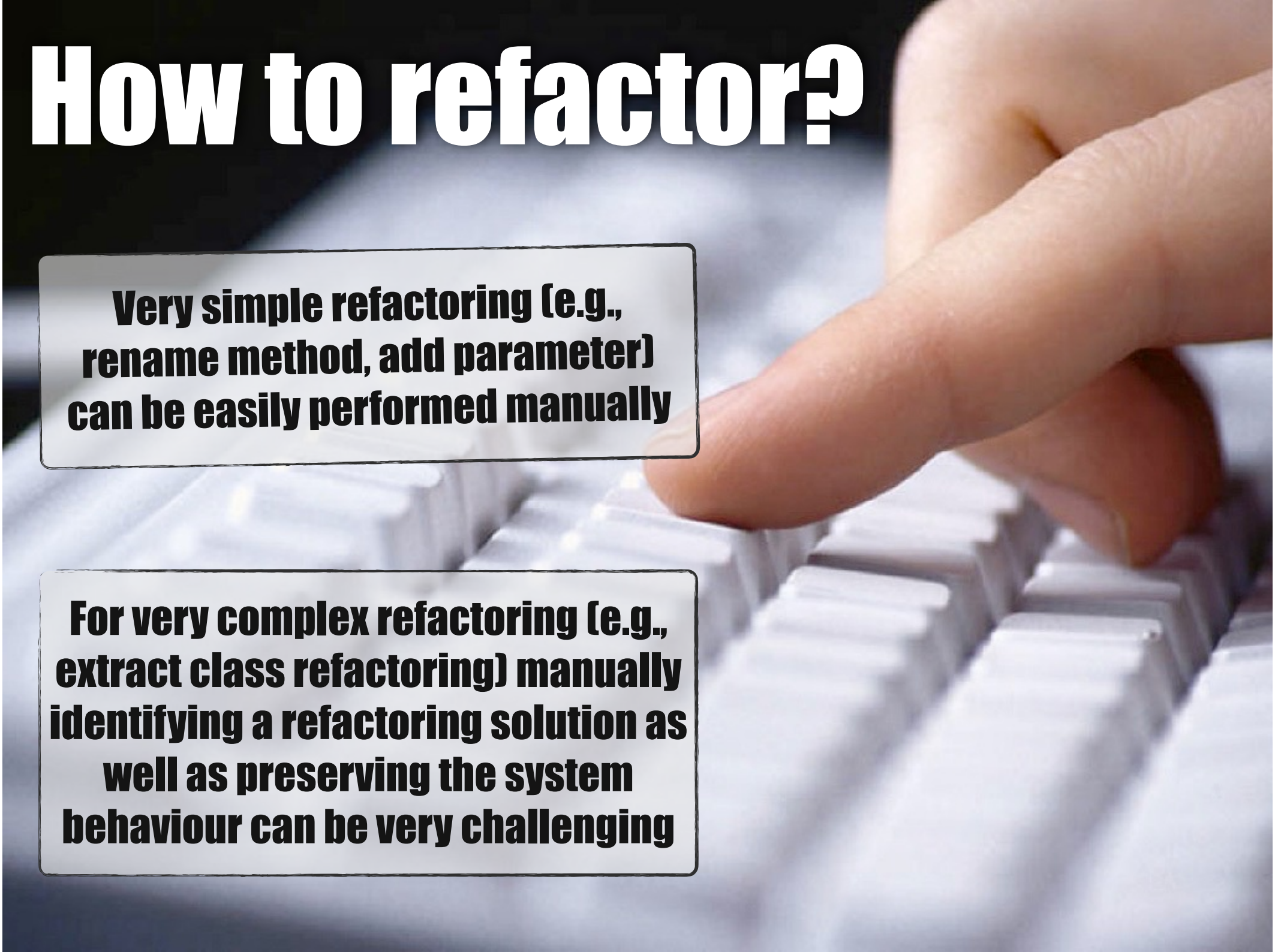
## Combining Structural, Textual, and historical information for smell detection

# The refactoring process



- Where to refactor
- How to refactor?
- Guarantee behaviour preservation
- Apply the refactoring
- Assess its effects on quality
- Consistently modify other artifacts

[Mens and Tourwé TSE 2004]


How to refactor?
Very simple refactoring (e.g., rename method, add parameter) can be easily performed manually
For very complex refactoring (e.g., extract class refactoring) manually identifying a refactoring solution as well as preserving the system behaviour can be very challenging

# How to refactor

Suppose that we have identified a Blob class.
Now, we want to refactor it through an Extract Class refactoring.
Let's say that our Blob is very small, 10 methods and 5 attributes.

## In how many possible ways can I refactor this Blob?

## 32,766

# How to refactor

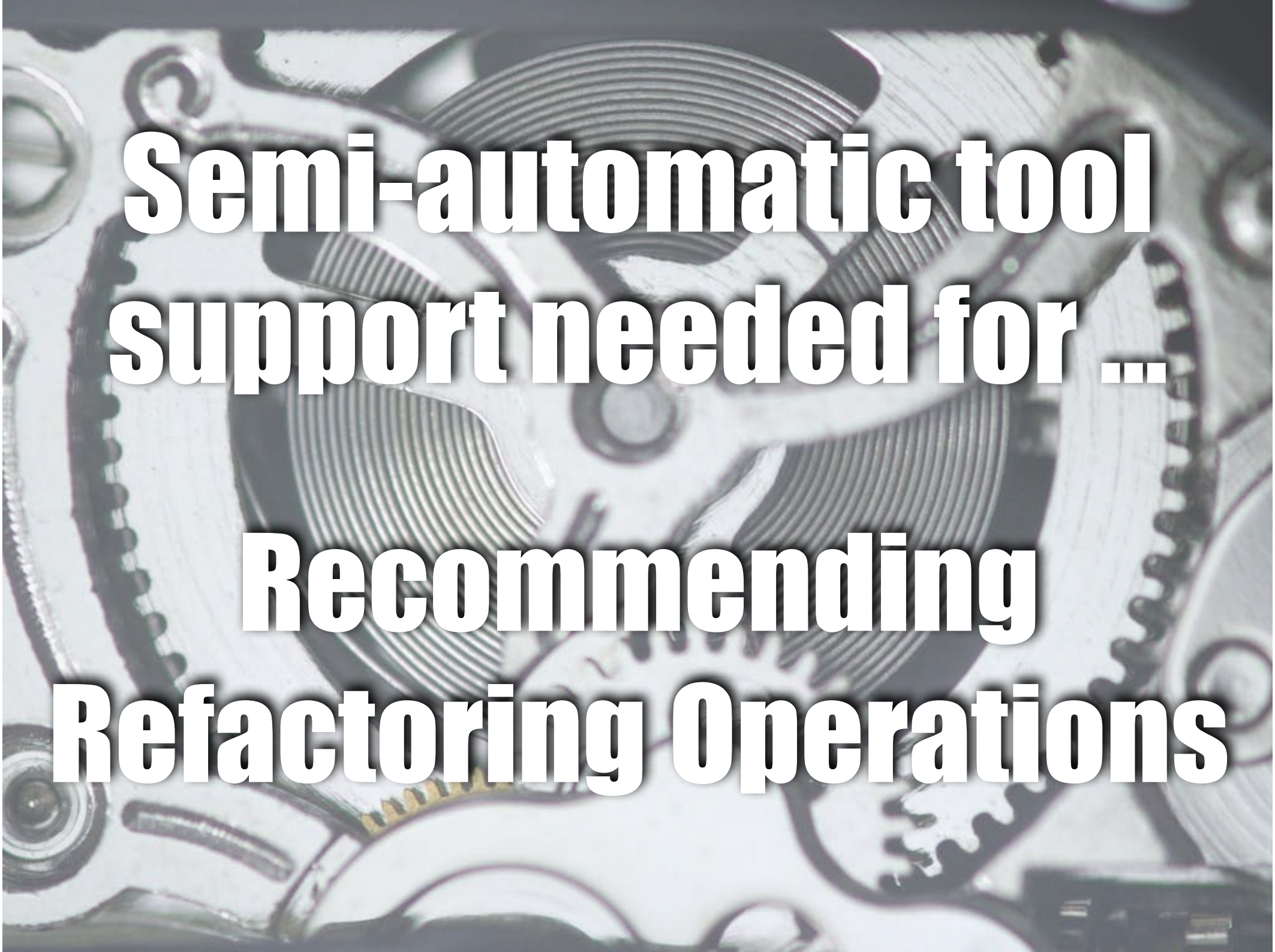
Suppose that we have identified a Blob class.
Now, we want to refactor it through an Extract Class refactoring.

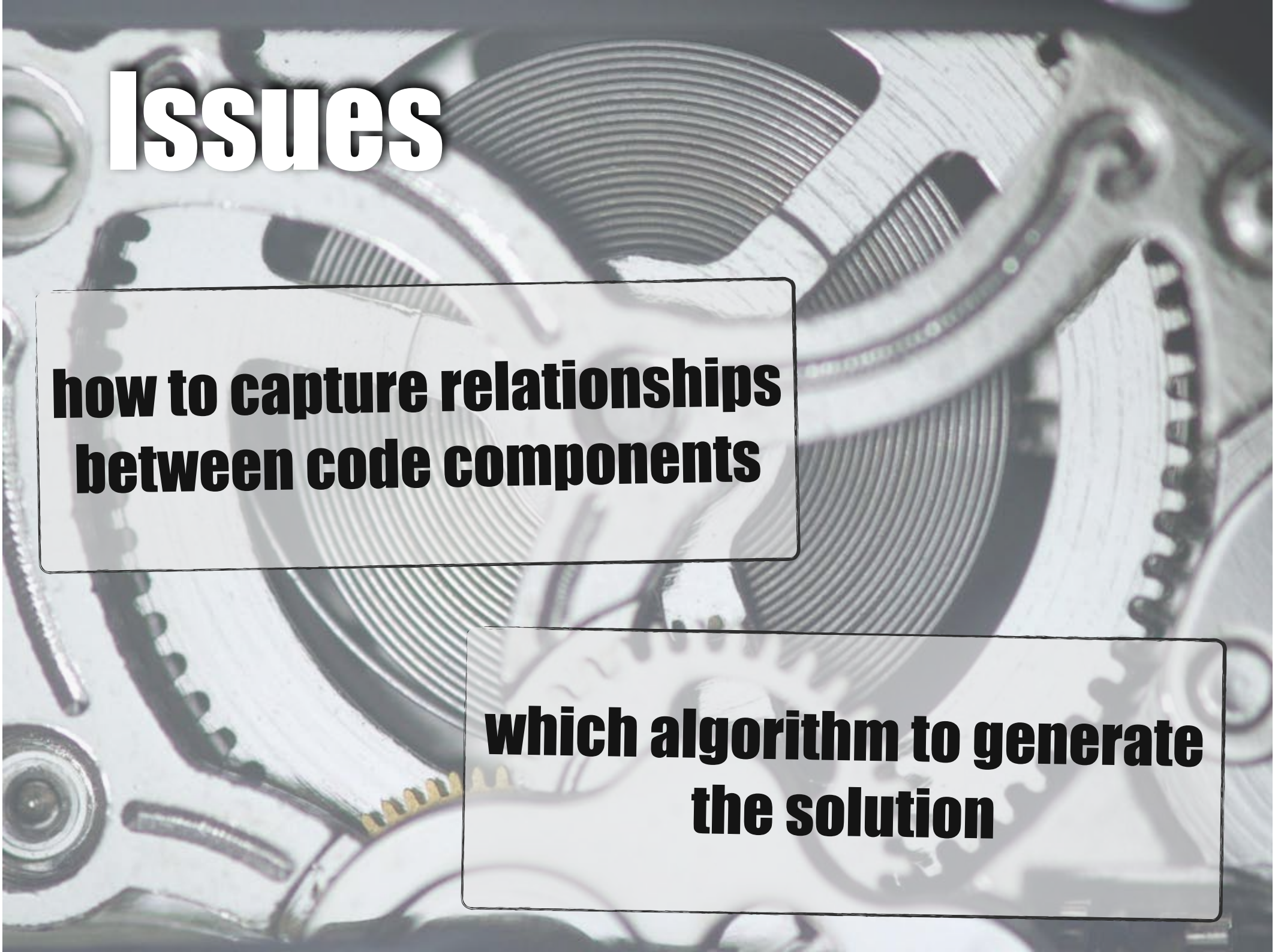
Let's take the example of a real Blob: 150 methods and 100 attributes.

## In how many possible ways can I refactor this Blob?

$$2^{250} \cong 10^{82}$$


$10^{82}$


Semi-automatic tool support needed for ...
Recommending Refactoring Operations


Issues
how to capture relationships between code components
which algorithm to generate the solution

# Capturing relationships between code components

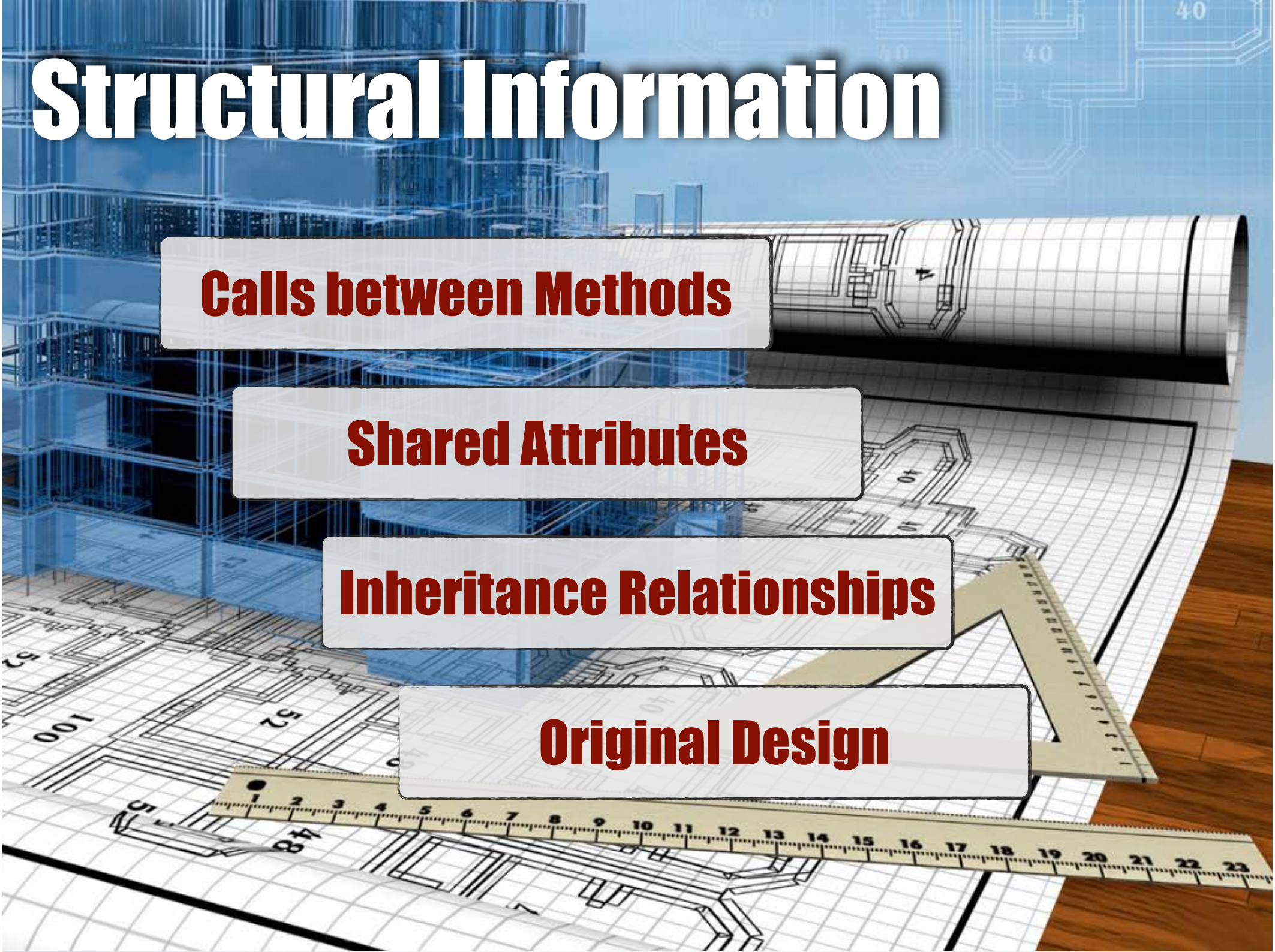
**Extract Class Refactoring Example**

**Relationships between methods and attributes of the Blob class to identify methods that are likely to implement the same (or similar) responsibilities, grouping them in a new extracted class together with the attributes they use**

# Possible Sources of Information

- Structural
- Dynamic
- Semantic
- Historical

# Structural Information



# Structural Information

Very easy to capture, always available, useful to support a wide range of refactorings

Do not tell the whole story.
Related code components not always have structural relationships among them

# Dynamic Information

Dependencies occurring during program execution

Dynamic information is more precise than structural information (on specific executions)

Dynamic information is more difficult to extract

# Semantic Information

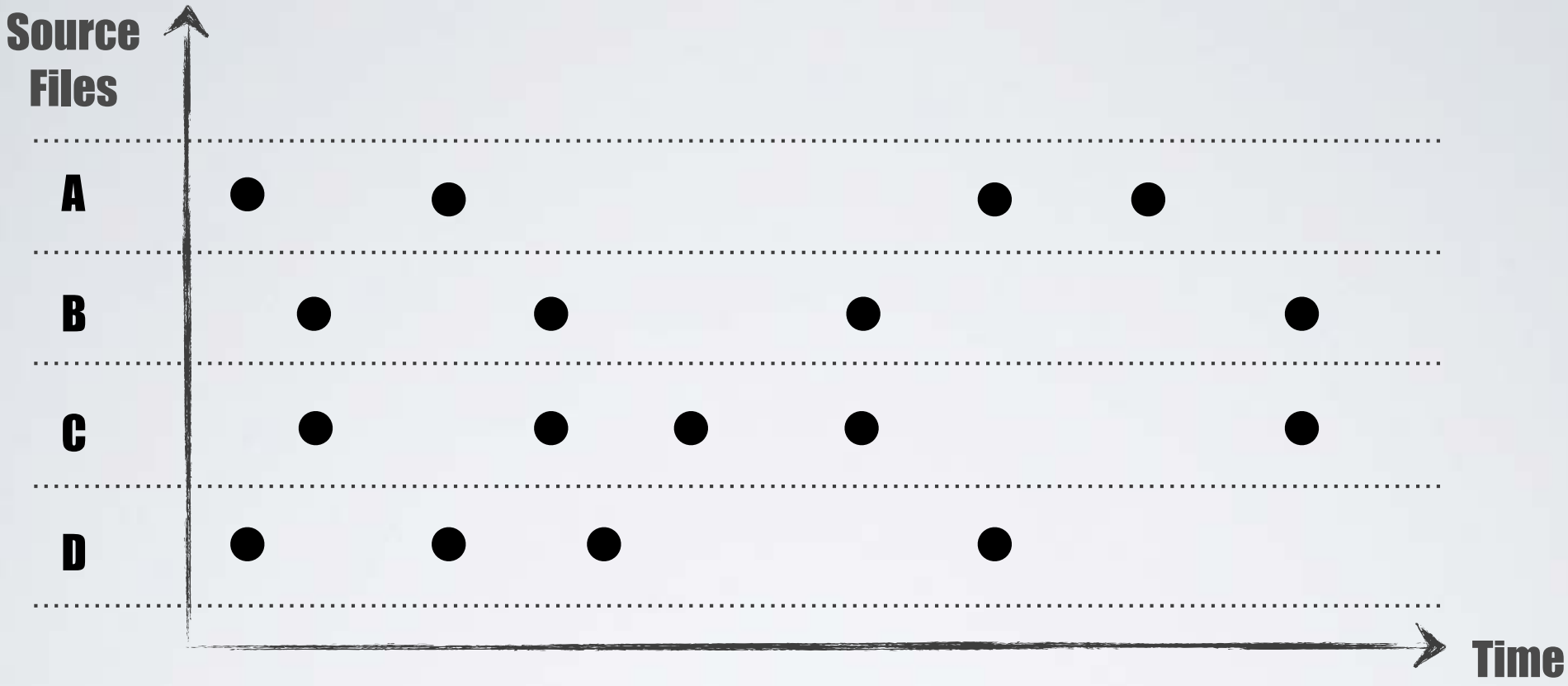
Textual similarity between code components

The conjecture: if developers used similar terms in comments and identifiers of two code components, it is likely they implement similar responsibilities

Very easy to capture, always available, useful to support a wide range of refactorings, but ...

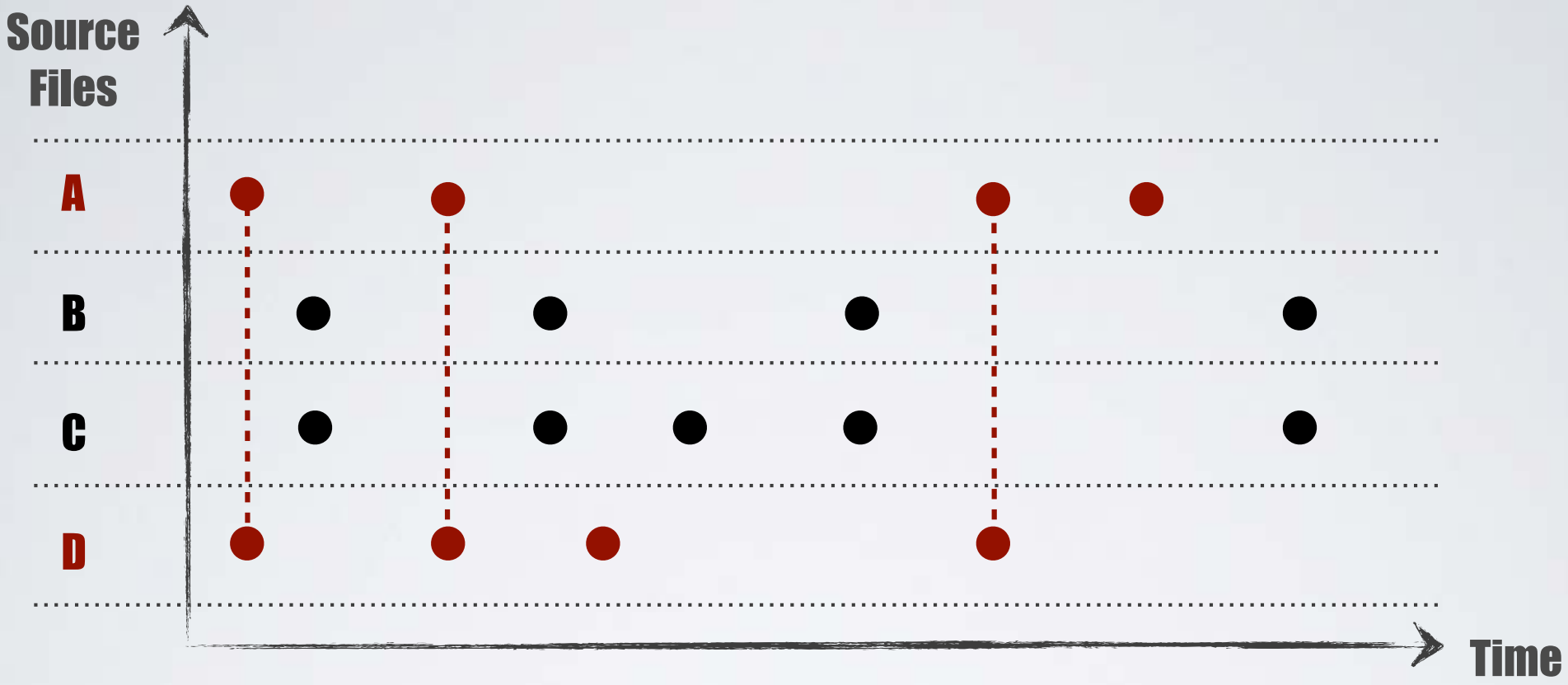
Strong assumption: developers consistently use terms in comments and identifiers

# Historical Information



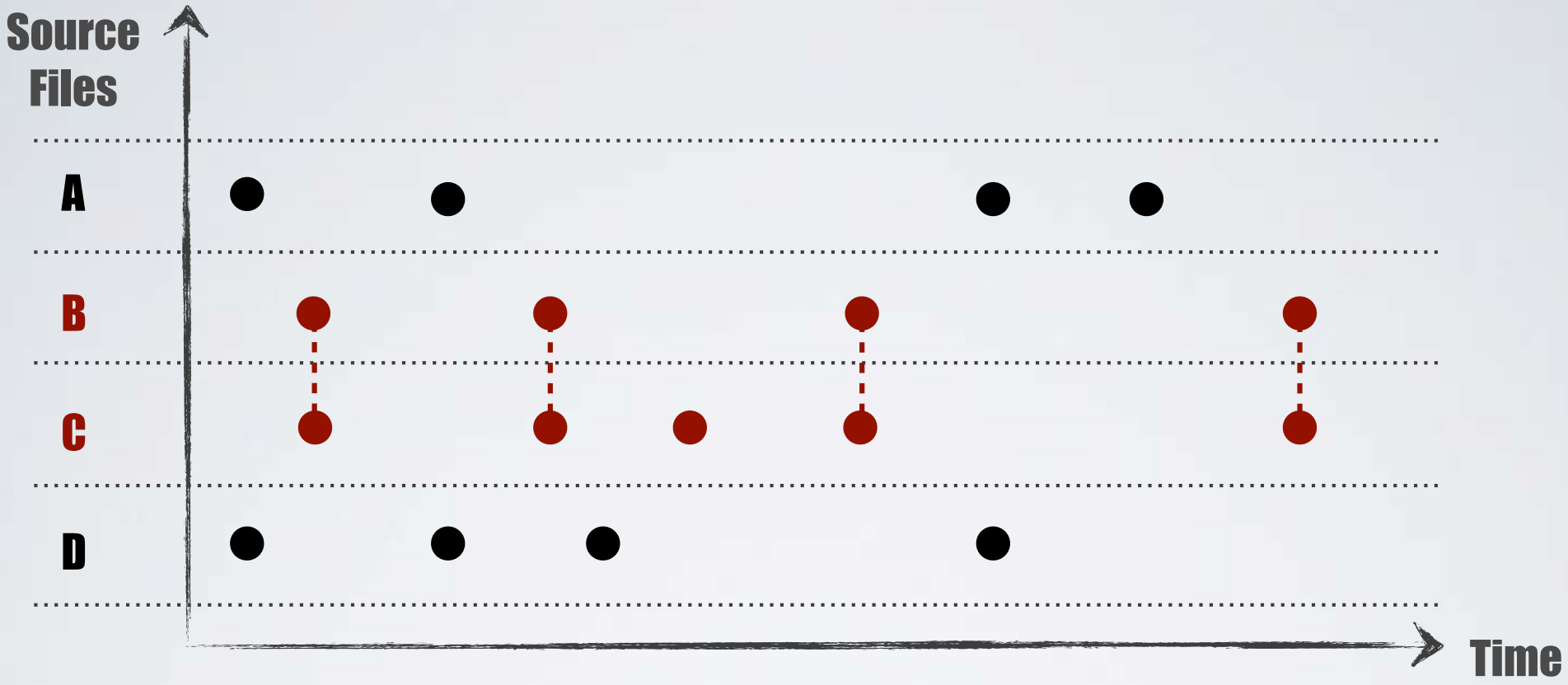

**Code components often changing together during software history are more likely to implement similar resposibilities**

# Historical Information



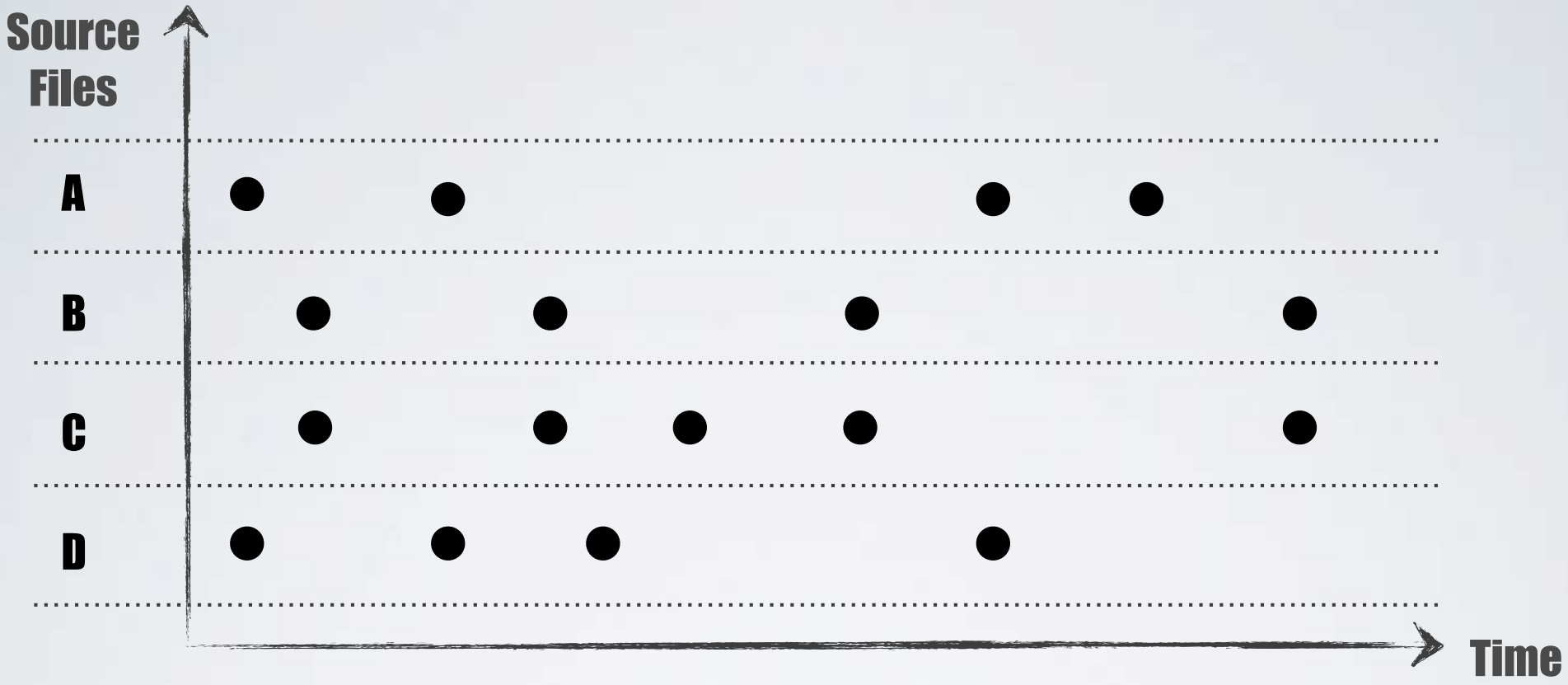

**Code components often changing together during software history are more likely to implement similar resposibilities**

# Historical Information




**Code components often changing together during software history are more likely to implement similar resposibilities**

# Historical Information




**Not always available !**

# Let's Summarize

Structural

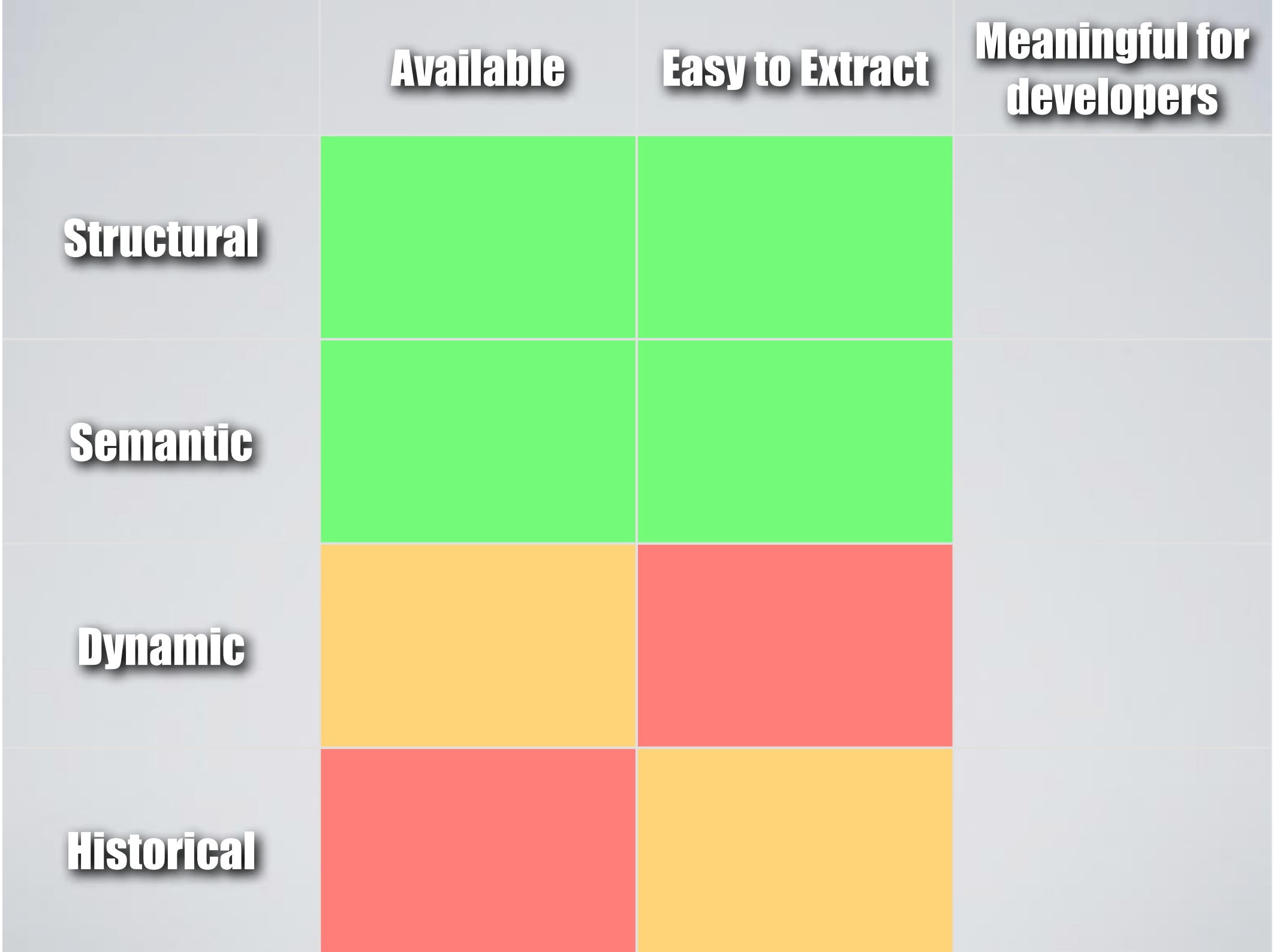
Semantic

Dynamic

Historical


Available
Structural
Semantic
Dynamic
Historical


Available
Easy to Extract
Structural
Semantic
Dynamic
Historical


Available
Easy to Extract
Meaningful for developers
Structural
Semantic
Dynamic
Historical


Available
Easy to Extract
Meaningful for developers
Structural
?
Semantic
?
Dynamic
?
Historical
?

# Which source of information better captures coupling between software entities as perceived by developers?

## Four types of coupling

- structural
- dynamic
- logical
- semantic

G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, A. De Lucia.
*An Empirical Study on the Developers Perception of Software Coupling.*
In Proceedings of the 35th **International Conference on Software Engineering (ICSE 2013)**.
10 pages, to appear.

## Object systems: ArgoUML, JHotDraw, JEdit

| 2 pairs high structural | 2 pairs low structural | 2 pairs high dynamic | 2 pairs low dynamic | 2 pairs high historical | 2 pairs low historical | 2 pairs high semantic | 2 pairs low semantic |
|---|---|---|---|---|---|---|---|

## Developers

**12** original developers of the three object systems (6 ArgoUML, 3 JHotDraw, 3 JEdit)

**64** external developers

## Procedure

For each system, for each of the 16 pairs of classes, we asked developers to provide (i) a score on a Likert scale ranging from 1 (two classes are not coupled) to 5 (two classes are strongly coupled), and (ii) an explanation for their score.

**Original developers just evaluated classes on the system they are involved in, external developers on all three object systems.**

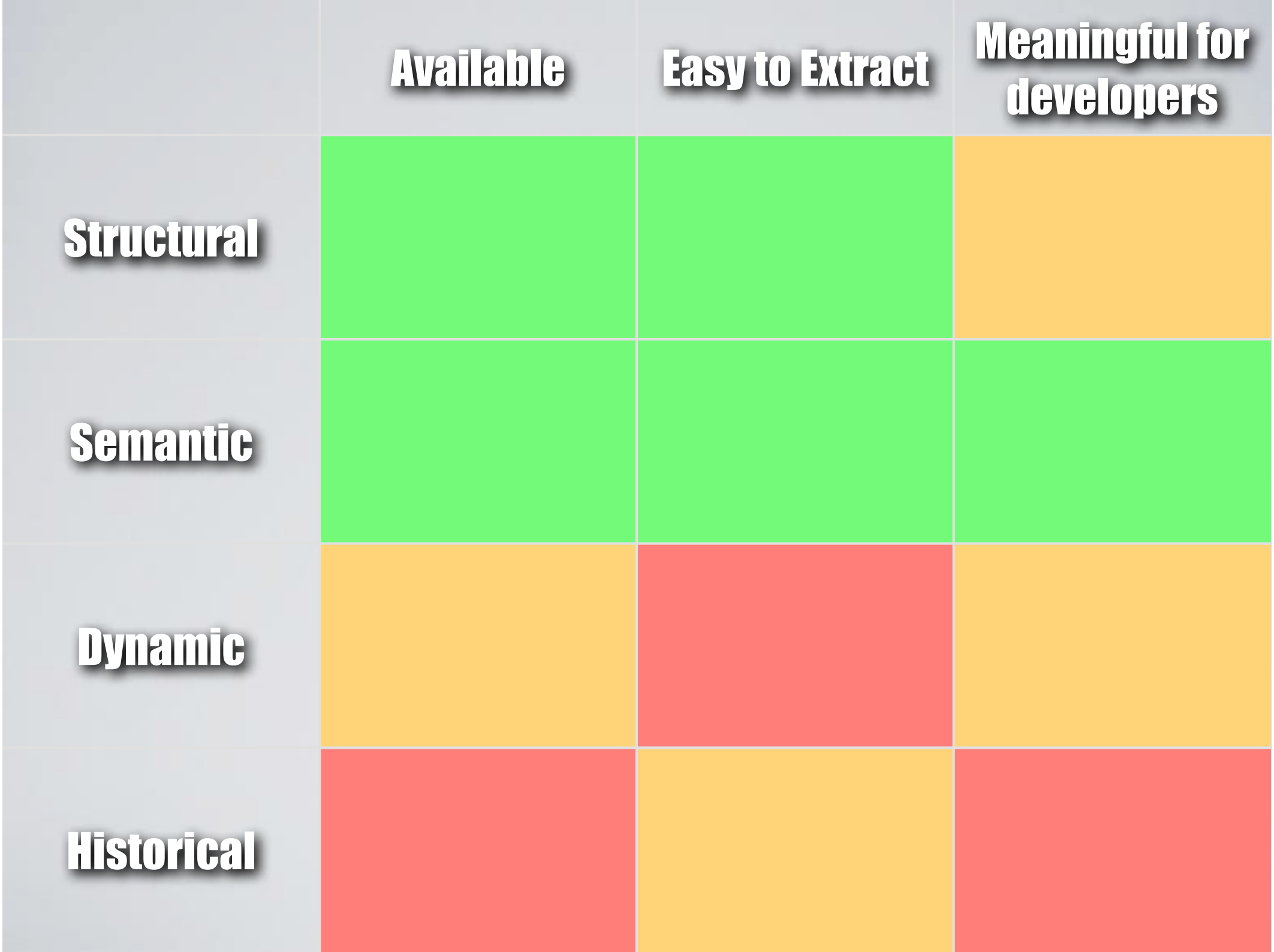
## Analysis

Boxplots and statistical test (Wilcoxon test)

# Results

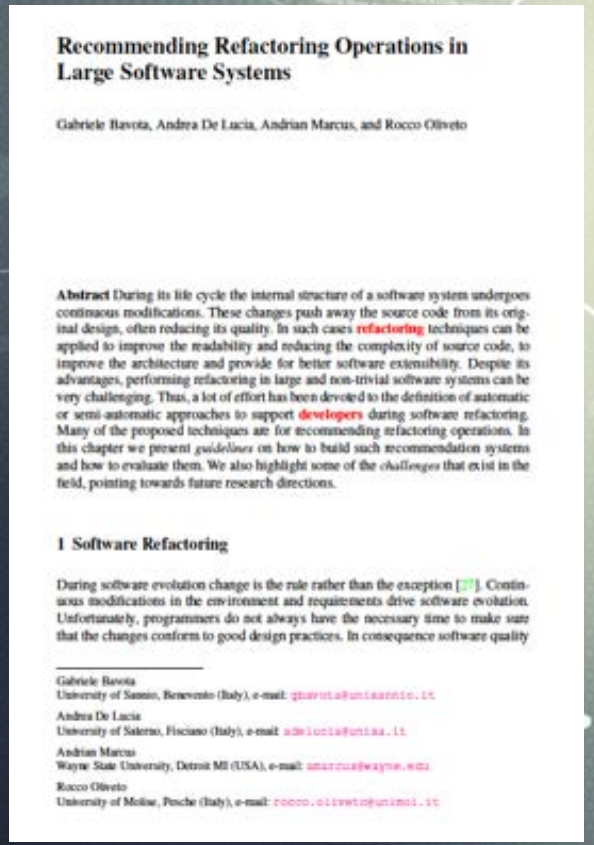

The **semantic** measure is the closest to the developers' perception of coupling.


Available
Easy to Extract
Meaningful for developers
Structural
?
Semantic
?
Dynamic
?
Historical
?


Available
Easy to Extract
Meaningful for developers
Structural
Semantic
Dynamic
Historical

# Which algorithm to generate the refactoring solution?

**The choice of the algorithm to be applied in order to identify a refactoring solution mainly depends on the refactoring operations that we are interested in supporting**



Recommending Refactoring Operations in Large Software Systems

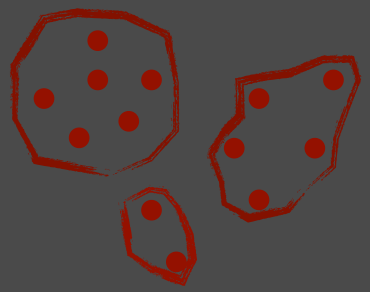
Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto

**Abstract** During its life cycle the internal structure of a software system undergoes continuous modifications. These changes push away the source code from its original design, often reducing its quality. In such cases **refactoring** techniques can be applied to improve the readability and reducing the complexity of source code, to improve the architecture and provide for better software extensibility. Despite its advantages, performing refactoring in large and non-trivial software systems can be very challenging. Thus, a lot of effort has been devoted to the definition of automatic or semi-automatic approaches to support **developers** during software refactoring. Many of the proposed techniques are for recommending refactoring operations. In this chapter we present *guidelines* on how to build such recommendation systems and how to evaluate them. We also highlight some of the *challenges* that exist in the field, pointing towards future research directions.

1 Software Refactoring

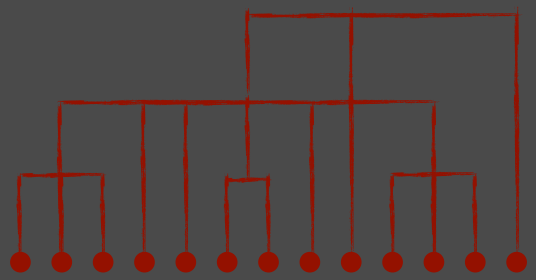
During software evolution change is the rule rather than the exception [27]. Continuous modifications in the environment and requirements drive software evolution. Unfortunately, programmers do not always have the necessary time to make sure that the changes conform to good design practices. In consequence software quality

Gabriele Bavota
University of Sannio, Benevento (Italy), e-mail: gbavota@unisannio.it
Andrea De Lucia
University of Salerno, Fisciano (Italy), e-mail: adelucia@unisa.it
Andrian Marcus
Wayne State University, Detroit MI (USA), e-mail: amarcus@wayne.edu
Rocco Oliveto
University of Molise, Pesche (Italy), e-mail: rocco.oliveto@unimol.it



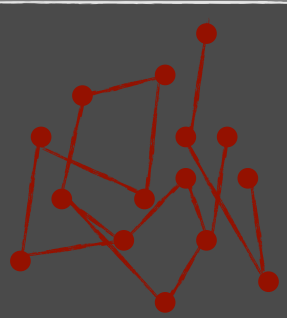
## Partitioning algorithms

The main problem to solve with these algorithms is related to the definition of the number of clusters to form. As example, partitioning algorithms like k-means explicitly require as input the number of clusters to be generated.



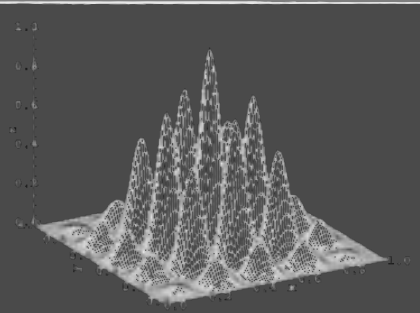
## Hierarchical algorithms

Finding the right level where to cut the dendrogram (i.e., determine the clusters), is a difficult problem that has to be solved by applying some heuristic . However, proposing alternative clustering solutions to the developers could also be seen as an advantage.



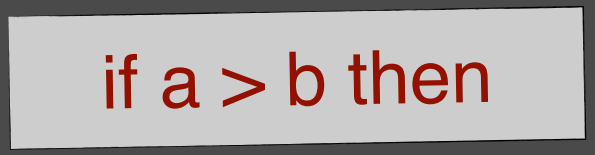
## Graph-Theory based

Represent the code components to be refactored as a weighted graph, where each node represents a component and the weight on the edge connecting two components their relationships. Graph-Theory algorithms can then be applied (e.g., MaxFlow-MinCut)



## Search-based algorithms

The task of refactoring is formulated as a search problem in the space of alternative designs. The alternative designs are generated applying a set of refactoring operations supported by the approach.




## Heuristic-based algorithms

When the problem faced by the refactoring operation is to move pieces of code to more appropriate place, a simple analysis of the dependencies between code components could be enough


The Journal of Systems and Software

Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures

**Supported Refactoring Operation:** Extract Class Refactoring

**Exploited Information:** Shared attributes, method calls, semantic similarity

**Algorithm:** Graph-based

G. Bavota, A. De Lucia, R. Oliveto.
*Identifying Extract Class Refactoring Opportunities Using Structural and Semantic Cohesion Measures.*
**Journal of Systems and Software**, 2011


Automating extract class refactoring: an improved method and its evaluation

**Supported Refactoring Operation:** Extract Class Refactoring

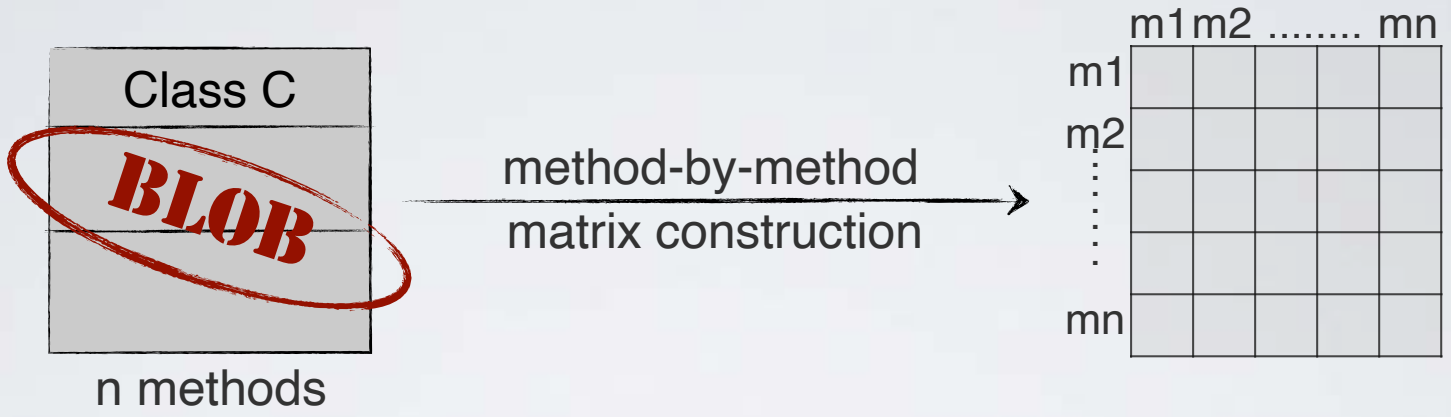
**Exploited Information:** Shared attributes, method calls, semantic similarity

**Algorithm:** Graph-based

**Details in next slides**

G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto.
*Automating Extract Class Refactoring: an Improved Method and its Evaluation.*
**Empirical Software Engineering (EMSE)**, 2014

# GRAPH theory-based ECR




## SSM

Structural Similarity between Methods

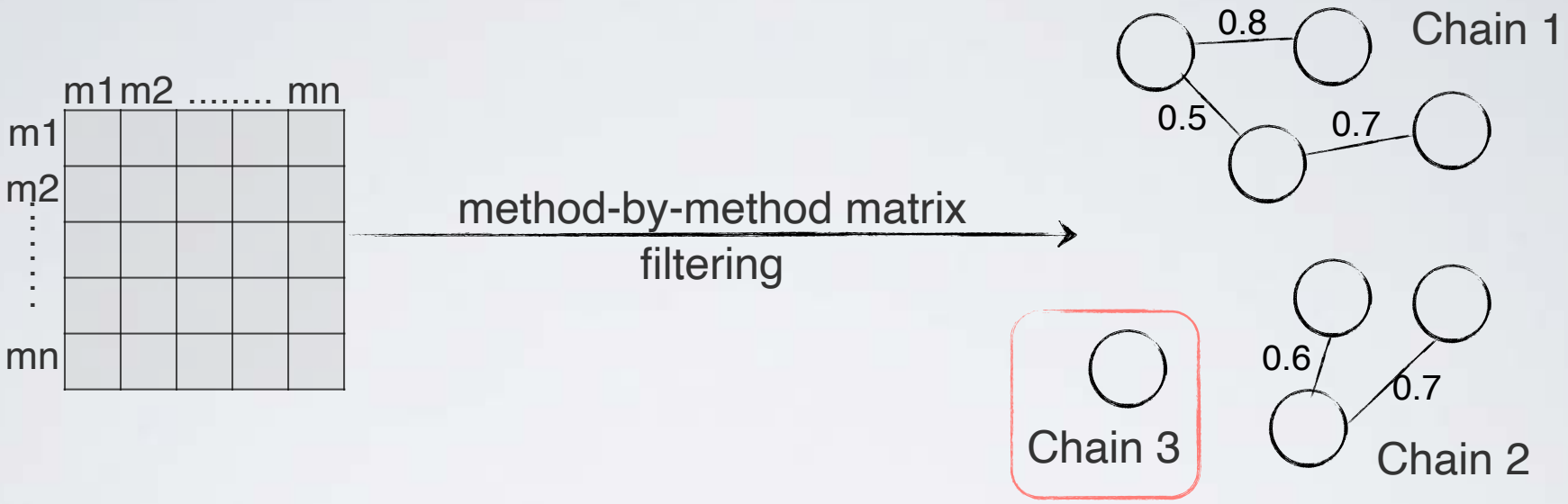
## CIM

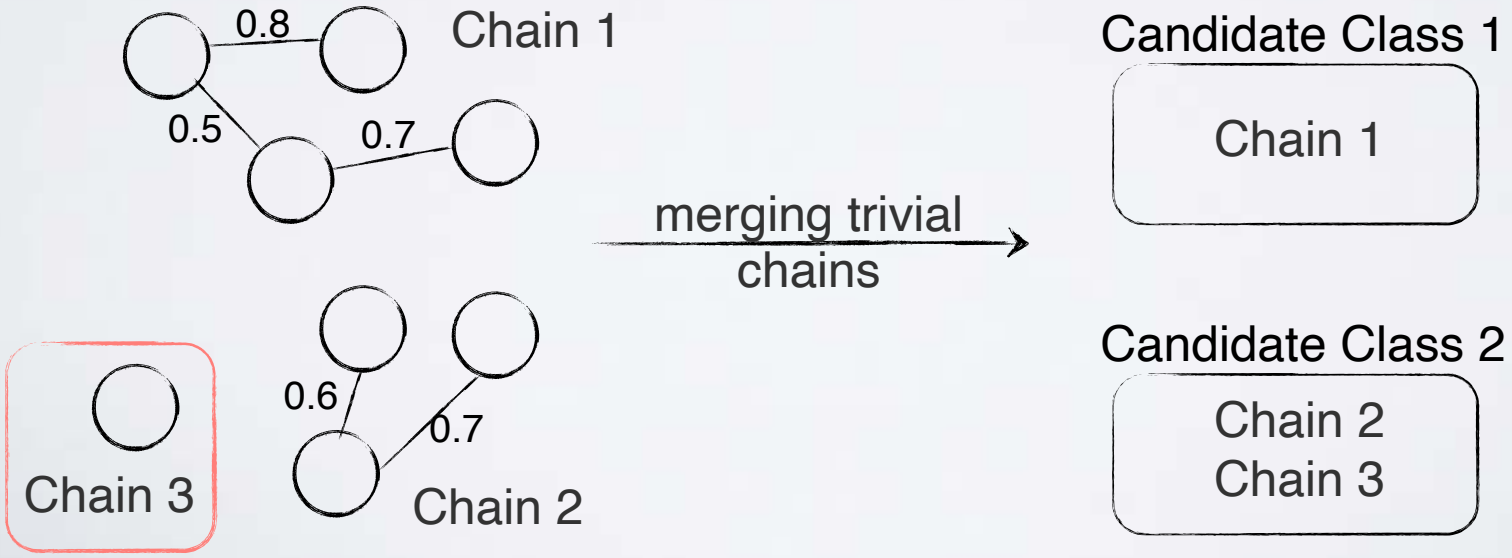
Call-based Interaction between Methods

## CSM

Conceptual Similarity between Methods

# GRAPH theory-based ECR



**Supported Refactoring Operation:** Extract Class Refactoring

**Exploited Information:** Shared attributes, method calls

**Algorithm:** Agglomerative Clustering

M. Fokaefs, N. Tsantalis, E. Stroulia, A. Chatzigeorgiou.
Identification and application of Extract Class refactorings in object-oriented systems
**ICSM 2009**

**Supported Refactoring Operation:** Extract Class Refactoring

**Exploited Information:** Shared attributes, method calls, inheritance relationships, original design

**Algorithm:** Search-based

O'Keeffe, M., O'Cinneide, M.
Search-based software maintenance.
**CSMR 2006**


Recommending Refactorings based on Team Co-Maintenance Patterns

**Supported Refactoring Operation:** Extract Class Refactoring

**Exploited Information:** Historical information about changes

**Algorithm:** Heuristic-based

G. Bavota, S. Panichella, N. Tsantalis, M. Di Penta, R. Oliveto, G. Canfora.
Recommending Refactorings based on Team Co-Maintenance Patterns
**ASE 2014**


In Medio Stat Virtus: Extract Class Refactoring through Nash Equilibria

**Supported Refactoring Operation:** Extract Class Refactoring

**Exploited Information:** Shared attributes, method calls, semantic similarity

**Algorithm:** Heuristic-based

G. Bavota, R. Oliveto, A. De Lucia, A. Marcus, Y.-G. Guéhéneuc, G. Antoniol.
In Medio Stat Virtus: Extract Class Refactoring through Nash Equilibria
**CSMR-WCRE 2014**

Using structural and semantic measures to improve software modularization

Abstract Changes during software evolution and poor design decisions often lead to packages that are hard to understand and maintain, because they usually group together classes with unrelated responsibilities. One way to improve such packages is to decompose them into smaller, more cohesive packages. The difficulty lies in the fact that most definitions and interpretations of cohesion are rather vague and the multitude of measures proposed by researchers usually capture only one aspect of cohesion. We propose a new technique for automatic re-modularization of packages, which uses structural and semantic measures to decompose a package into smaller, more cohesive ones. The paper presents the new approach as well as an empirical study, which evaluates the decompositions proposed by the new technique. The results of the evaluation indicate that the decomposed packages have better cohesion without a deterioration of coupling and the re-modularizations proposed by the tool are also meaningful from a functional point of view.

Keywords Software re-modularization · Information-flow-based coupling · Conceptual coupling between classes · Empirical studies

**Supported Refactoring Operation:** Extract Package Refactoring

**Exploited Information:** Method calls, semantic similarity

**Algorithm:** Graph-based

G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto.
*Using structural and semantic measures to improve software modularization.*
**Empirical Software Engineering (EMSE)**, 2013

Improving Software Modularization via Automated Analysis of Latent Topics and Dependencies

GABRIELE BAVOTA, MALCOM GETHERS, ROCCO OLIVETO, DENYS POSHYVANYK, ANDREA DE LUCIA

**Supported Refactoring Operation:** Move Class Refactoring

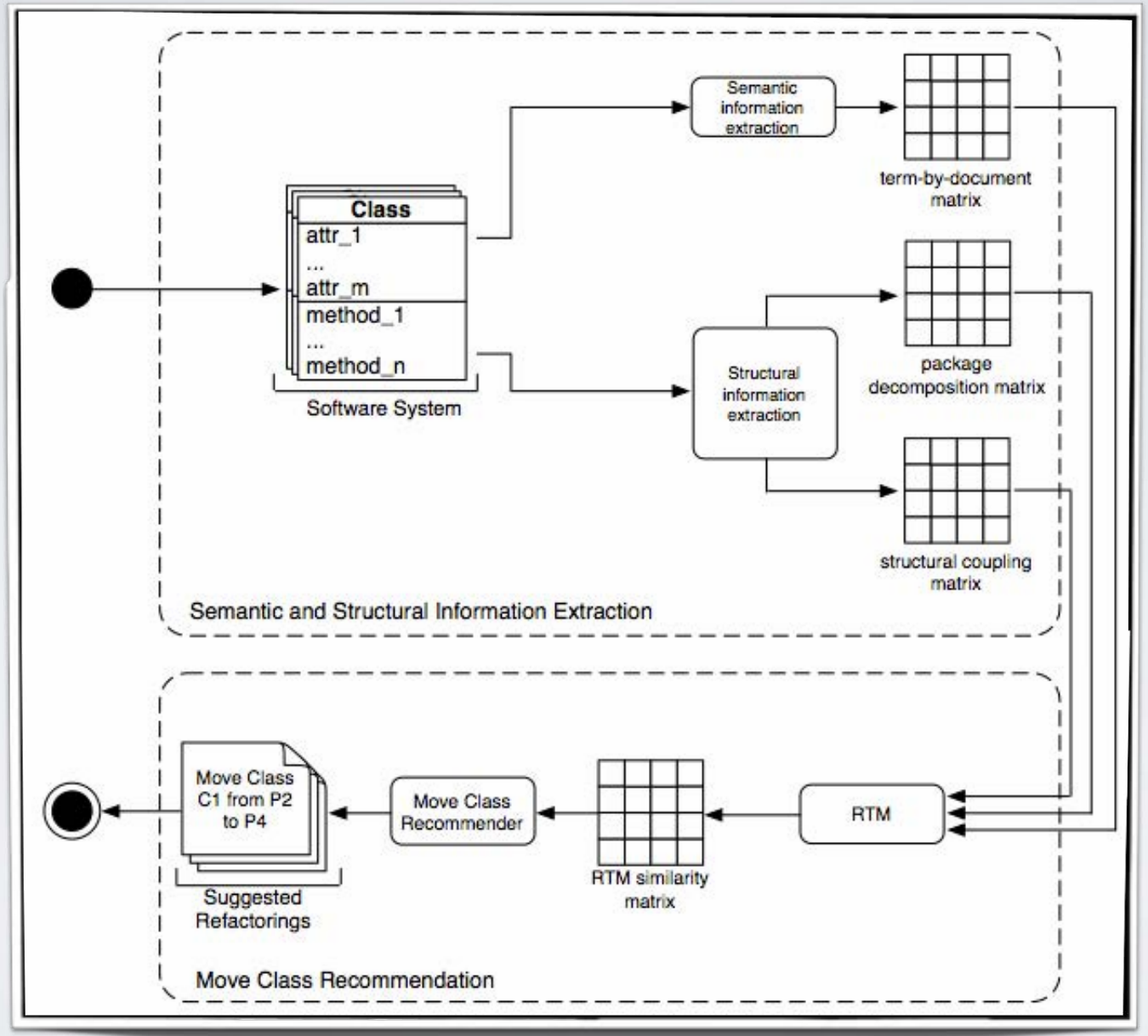
**Exploited Information:** Method calls, original design, semantic similarity

**Algorithm:** Heuristic-based

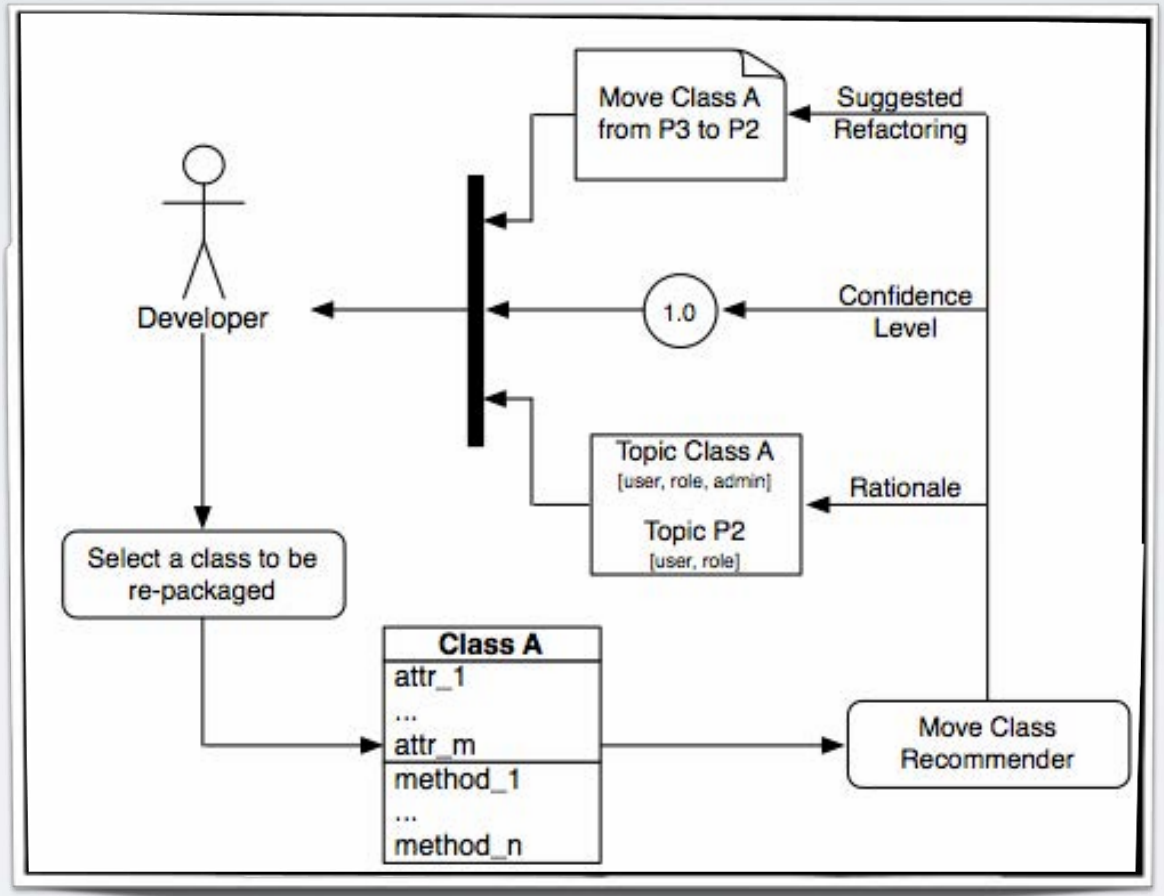
**Details in next slides**

G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, A. De Lucia.
Improving Software Modularization via Automated Analysis of Latent Topics and Dependencies.
**Transactions on Software Engineering and Methodology (TOSEM)**, 2014

# R3: Move Class Refactoring via RTM



# R3: feedbacks provided to developers



Confidence Level

0.69

MOVE class ActionExportProfileXMI implementing the topics [profile, model, url]
FROM its package org.argouml.ui.explorer grouping the topics [tree, node, explor]
TO the package org.argouml.profile grouping the topics [profile, ocl, model]


Identification of Move Method Refactoring Opportunities

**Supported Refactoring Operation:** Move Method Refactoring

**Exploited Information:** Shared attributes, method calls, original design

**Algorithm:** Heuristic-based

Nikolaos Tsantalis, Alexander Chatzigeorgiou.
Identification of Move Method Refactoring Opportunities.
**Transactions on Software Engineering (TSE)**, 2009


Methodbook: Recommending Move Method Refactorings via Relational Topic Models

**Supported Refactoring Operation:** Move Method Refactoring

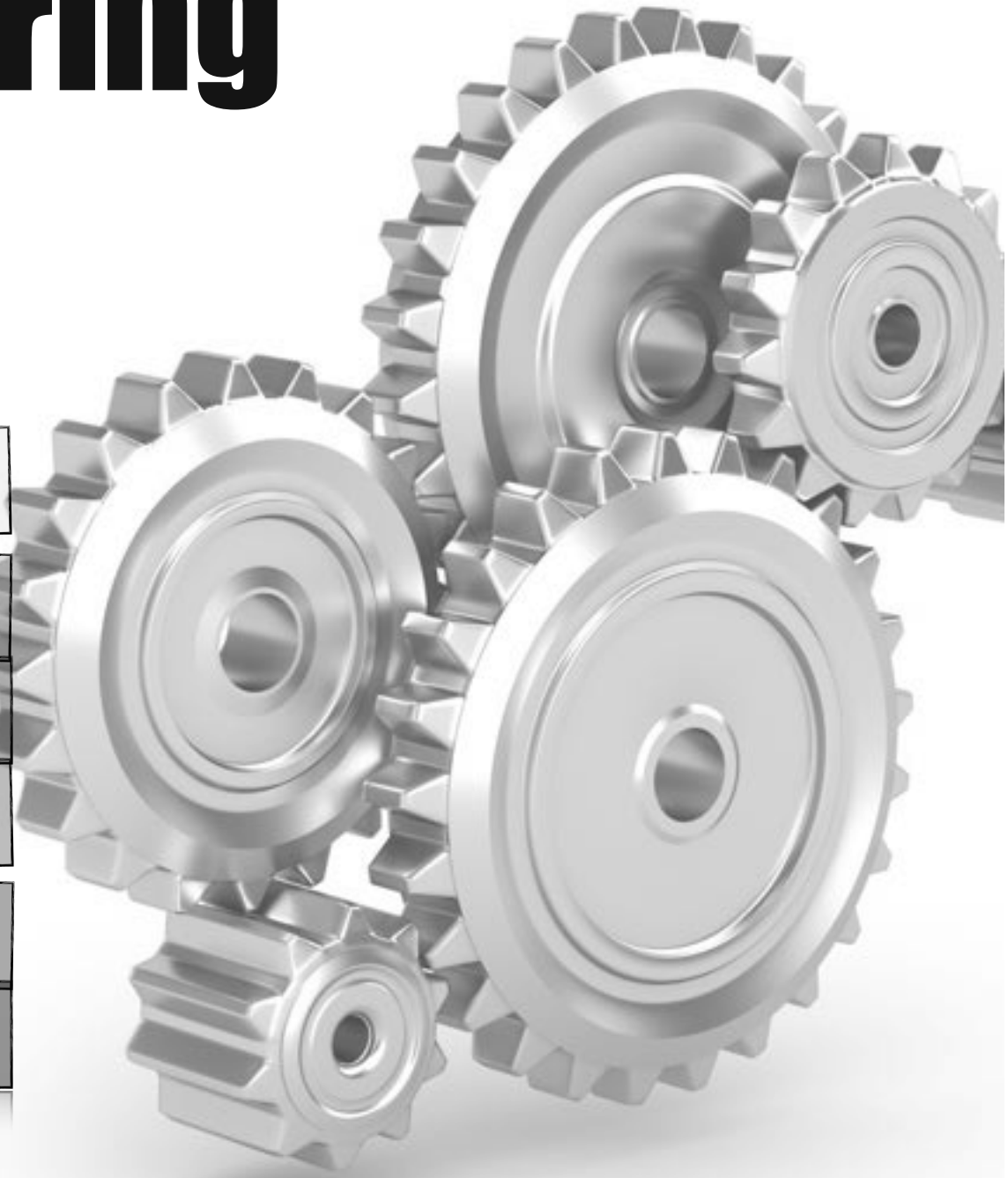
**Exploited Information:** Shared attributes, method calls, original design, semantic similarity

**Algorithm:** Heuristic-based

G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, A. De Lucia.
Methodbook: Recommending Move Method Refactorings via Relational Topic Models
**Transactions on Software Engineering (TSE)**, 2014

# The refactoring process

- Where to refactor
- How to refactor?
- **Guarantee behaviour preservation**
- Apply the refactoring
- Assess its effects on quality
- Consistently modify other artifacts



[Mens and Tourwé TSE 2004]

# Guarantee behaviour preservation

Manually feasible for very simple refactoring (e.g., rename method, add parameter)

Time consuming and error prone if manually performed for complex refactorings

## Bavota et. al - SCAM 2012

**15%** of refactoring operations induces bugs

**40%** for more complex refactorings (e.g., extract subclass)

## Kim et. al - FSE 2012

Survey performed with 328 Microsoft engineers

**50%** of developers do not define refactoring as a "behaviour preserving activity"

**51%** manually perform refactoring

**77%** feel bug introduction as main risk during refactoring

## Murphy-Hill et. al - TSE 2011

**90%** of refactoring activities are performed manually

**Automated Testing of Refactoring Engines**

Brett Daniel Danny Dig Kely Garcia Darko Marinov
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
(bdaniel3, dig, kgarcia2, marinov)@cs.uiuc.edu

**ABSTRACT**

Refactorings are behavior-preserving program transformations that improve the design of a program. Refactoring engines are tools that automate the application of refactorings: first the user chooses a refactoring to apply, then the engine checks if the transformation is safe, and if so, transforms the program. Refactoring engines are a key component of modern IDEs, and programmers rely on them to perform refactorings. A bug in the refactoring engine can have severe consequences as it can erroneously change large bodies of source code.

We present a technique for automated testing of refactoring engines. Test inputs for refactoring engines are programs. The core of our technique is a framework for iterative generation of structurally complex test inputs. We instantiate the framework to generate abstract syntax trees that represent Java programs. We also create several kinds of oracles to automatically check that the refactoring engine transformed the generated program correctly. We have applied our technique to testing Eclipse and NetBeans, two popular open-source IDEs for Java, and we have exposed 21 new bugs in Eclipse and 24 new bugs in NetBeans.

**Categories and Subject Descriptors:** D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*object-oriented programming*

**General Terms:** Verification

**Keywords:** Automated testing, bounded-exhaustive testing, imperative generators, test data generation, refactoring engines

**1. INTRODUCTION**

Refactoring [9] is a disciplined technique of applying behavior-preserving transformations to a program with the intent of improving its design. Examples of refactorings include renaming a program element to better convey its meaning, replacing field references with calls to accessor methods, splitting large classes, moving methods to different classes, or extracting duplicated code into a new method. Each refactoring has a name, a set of preconditions, and a set of specific transformations to perform [19].

*Refactoring engines* are tools that automate the application of refactorings. The programmer need only select which refactoring to apply, and the engine will automatically check the preconditions and apply the transformations across the entire program if the preconditions are satisfied. Refactoring is gaining popularity, as evidenced by the inclusion of refactoring engines in modern IDEs such as Eclipse (http://www.eclipse.org) or NetBeans (http://www.netbeans.org) for Java. Refactoring is also a key practice of agile software development methodologies, such as eXtreme Programming [4], whose success prompts even more developers to use refactoring engines on a regular basis. Indeed, the common wisdom views the use of refactoring engines as one of the safest ways of transforming a program, since manual refactoring is error-prone.

It is important that refactoring engines be reliable—a bug in a refactoring engine can silently introduce bugs in the refactored program and lead to difficult debugging sessions. If the original program compiles but the refactored program does not, the refactoring is obviously incorrect and can be easily undone. However, if the refactoring engine erroneously produces a refactored program that compiles but does not preserve the semantics of the original program, this can have severe consequences.

Since refactoring engines are very complex and must be reliable, developers of refactoring engines have invested heavily in testing. For example, Eclipse version 3.2 has over 2,600 unit tests for refactorings (publicly available from the Eclipse CVS repository). Conventionally, testing a refactoring engine involves creating input programs by hand along with their expected outputs, each of which is either a refactored program or an expected precondition failure. The developers then execute these tests automatically with a tool such as JUnit [10]. Writing such tests manually is tedious and results in incomplete test suites, potentially leaving many hidden bugs in refactoring engines.

We present a technique that automates testing of refactoring engines, both generation of test inputs and checking of test outputs. The core of our technique is a general framework for iterative generation of structurally complex test inputs. We instantiate the general framework in a library called ASTGen. ASTGen allows developers to write *imperative generators* whose executions produce input programs for refactoring engines. More precisely, ASTGen offers a library of generic, reusable, and composable generators that produce abstract syntax trees (ASTs). Using ASTGen, a developer can focus on the creative aspects of testing rather than the mechanical production of test inputs. Instead of manually writing input programs, a developer writes a generator whose execution produces thousands of programs with structural properties that are relevant for the specific refactoring being tested. For example, to test the RenameField refactoring, the input program should have a



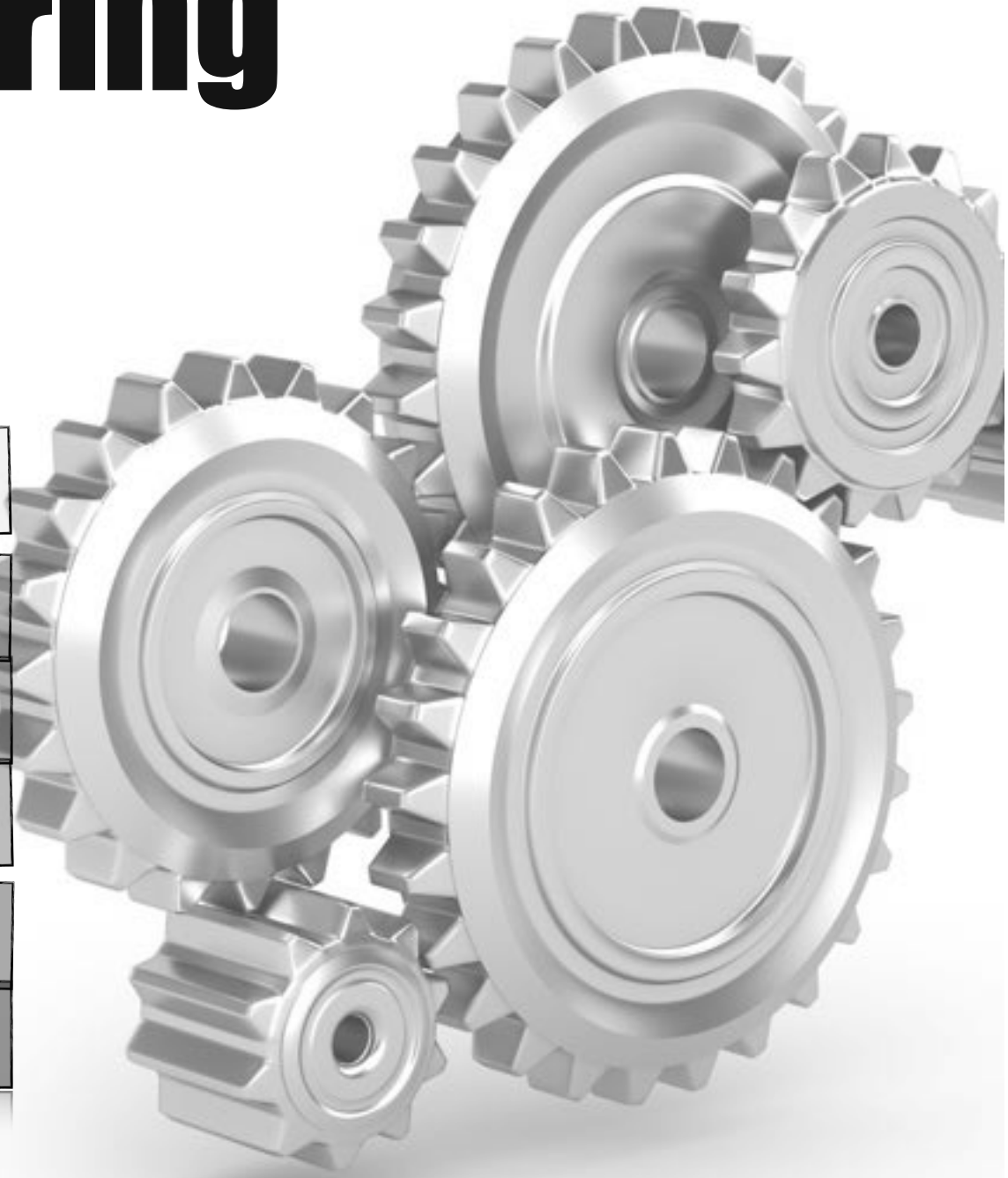
**Even automated refactoring performed by Integrated Development Environments could be fault-prone**

**45 previously unreported bugs in Eclipse and NetBeans**

[Daniel et al. ESEC/FSE 2007]


Stepping Stones over the Refactoring Rubicon
Lightweight Language Extensions to Easily Realise Refactorings
Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor
Programming Tools Group, University of Oxford, UK
Abstract. Refactoring tools allow the programmer to pretend they are
Differential Precondition Checking:
A Lightweight, Reusable Analysis
for Refactoring Tools
Formalising Refactorings with Graph Transformations*
Niels Van Eetvelde, Dirk Janssens, Serge Demeyer
Department of Mathematics and Computer Science
Universiteit Antwerpen
No approaches fully tested,
implemented in tools, able to support a
wide range of refactoring operations

# The refactoring process



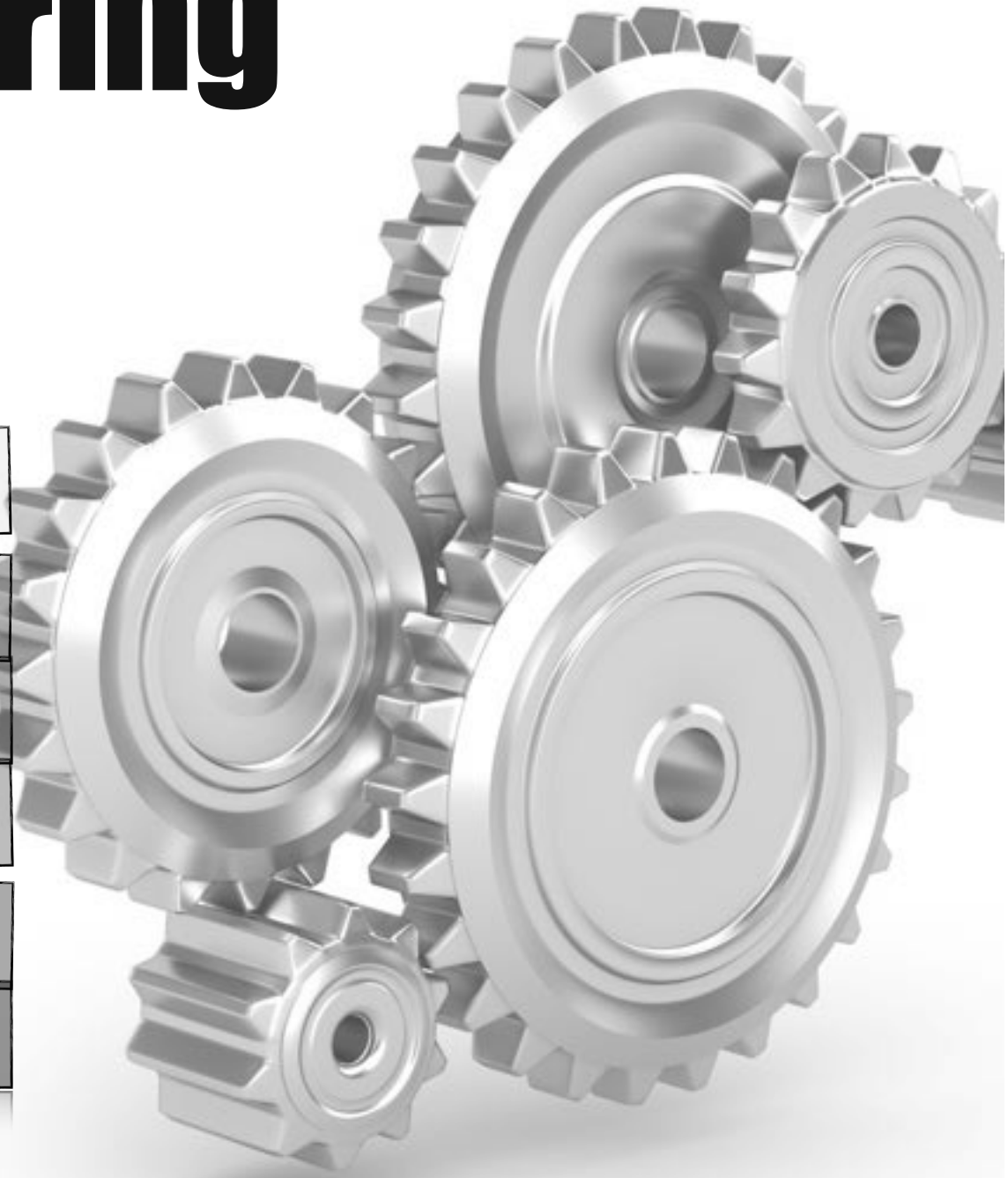
- Where to refactor
- How to refactor?
- Guarantee behaviour preservation
- Apply the refactoring
- Assess its effects on quality
- Consistently modify other artifacts

[Mens and Tourwé TSE 2004]

# How about Regression Testing ?

# The refactoring process

- Where to refactor
- How to refactor?
- Guarantee behaviour preservation
- Apply the refactoring
- Assess its effects on quality
- Consistently modify other artifacts



[Mens and Tourwé TSE 2004]

# Maintaining consistency of refactored software

Software development involves a wide range of software artifacts such as requirements specifications, software architectures, design models, source code, documentation, test suites, and so on. If we refactor any of these software artifacts, we need mechanisms to maintain their consistency.

## POSSIBLE SOLUTIONS

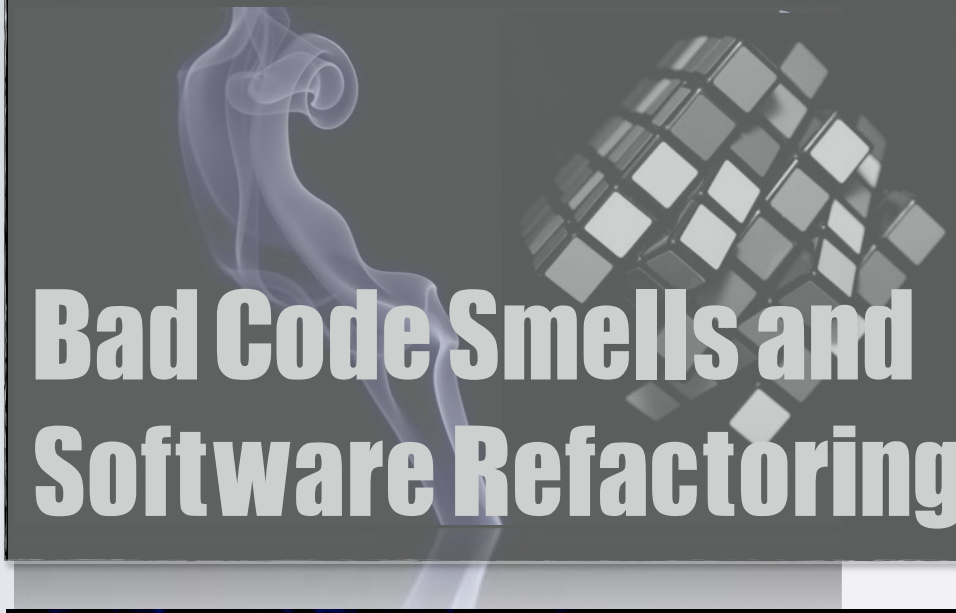
Explicitly maintain traceability between software artifacts

Recoverying traceability when needed, see e.g., Quesef et al. “Recovering test-to-code traceability using slicing and textual analysis”, JSS 2014

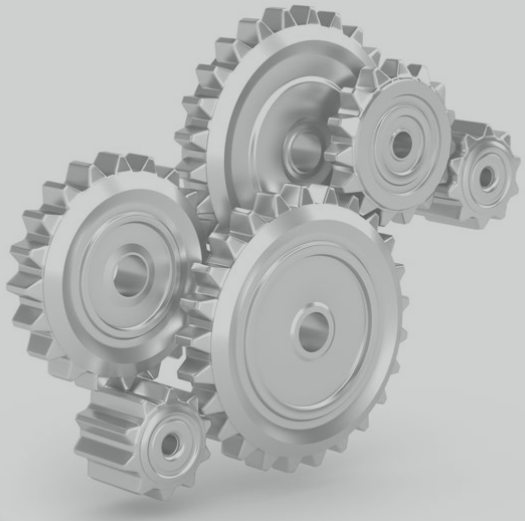
Change propagation techniques, see e.g., V. Rajlich, “A model for change propagation based on graph rewriting”, ICSM 1997

# Part I

## Bad Code Smells and Software Refactoring

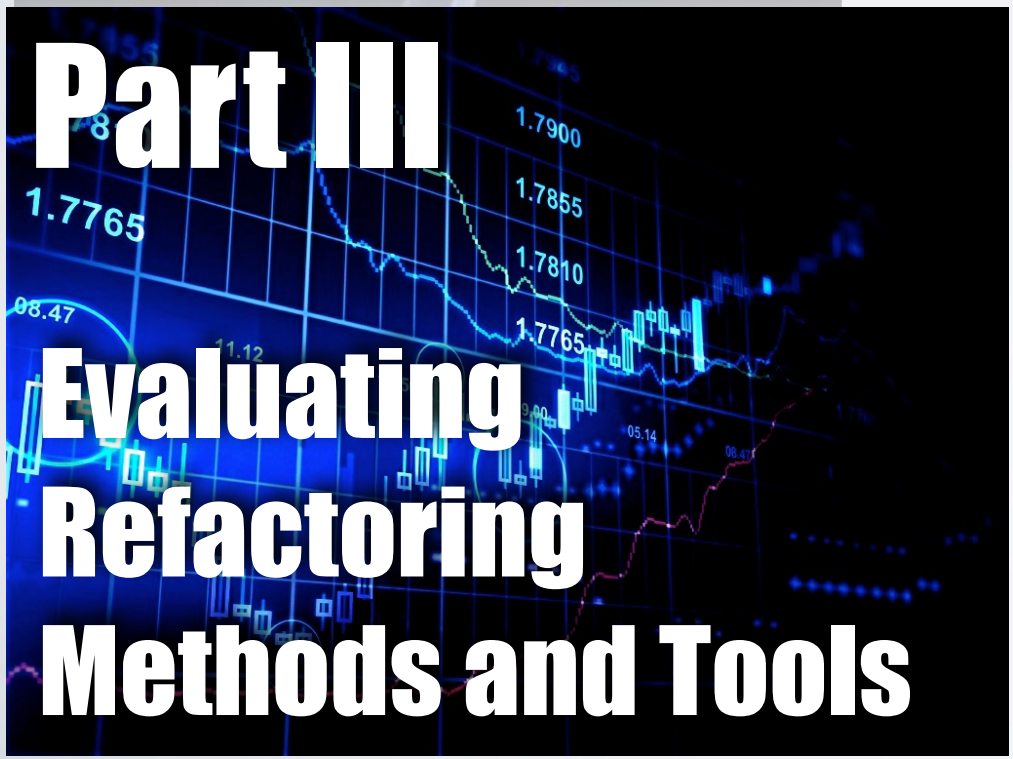


# Part II



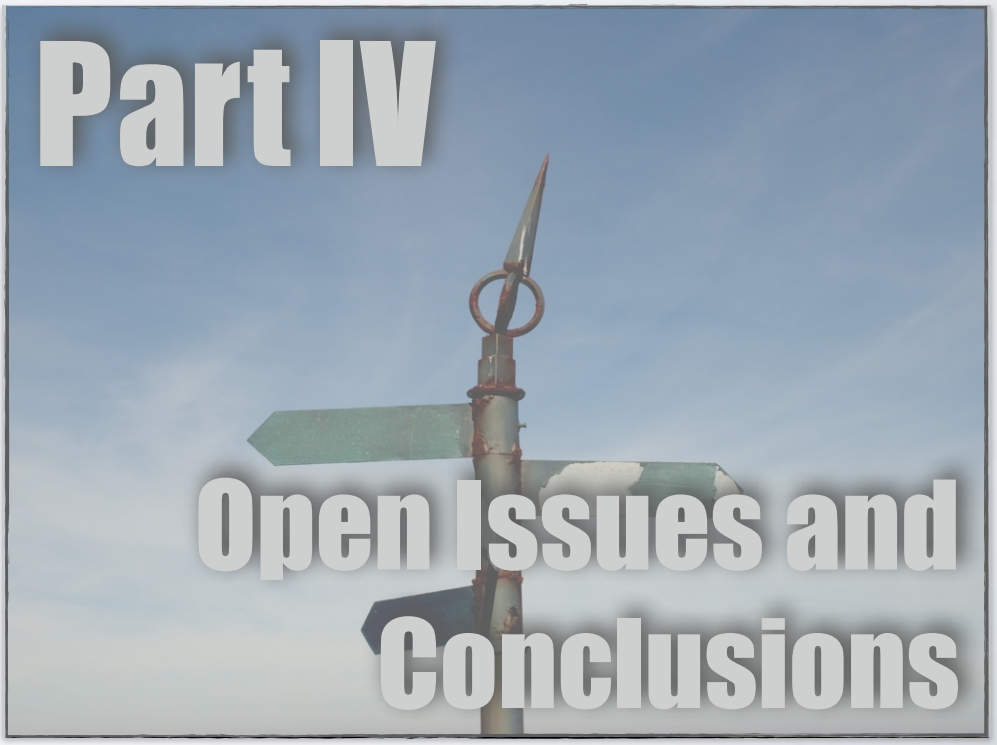
## The refactoring process

# Part III




## Evaluating Refactoring Methods and Tools

# Part IV

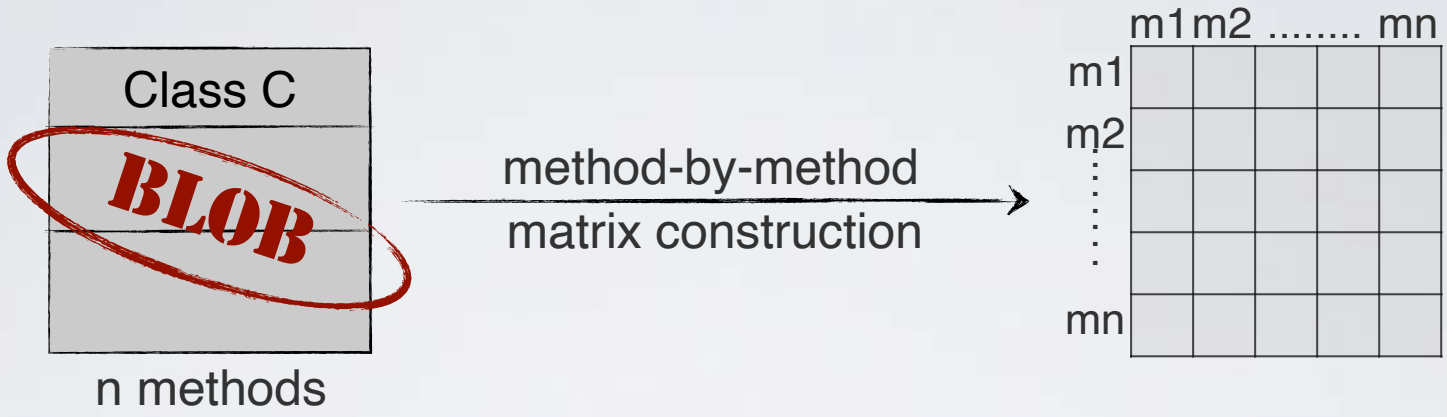


## Open Issues and Conclusions

# Extract Class Refactoring

G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto.
*Automating Extract Class Refactoring: an Improved Method and its Evaluation.*
**Empirical Software Engineering (EMSE), 2014**

# GRAPH theory-based




## SSM

Structural Similarity between Methods

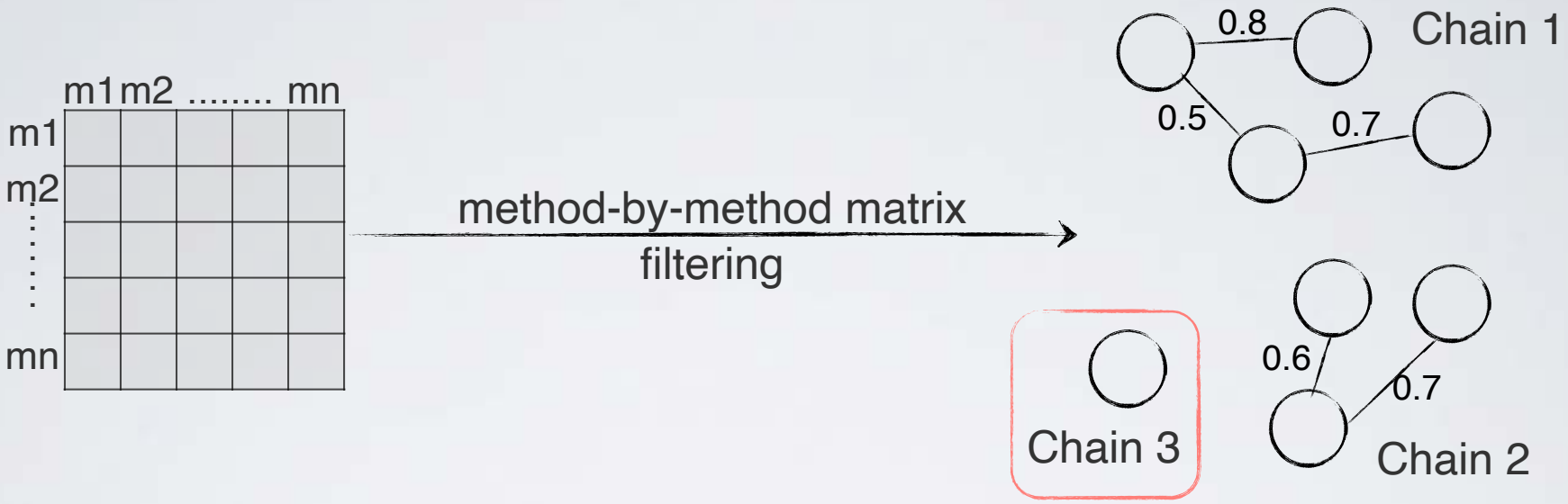
## CIM

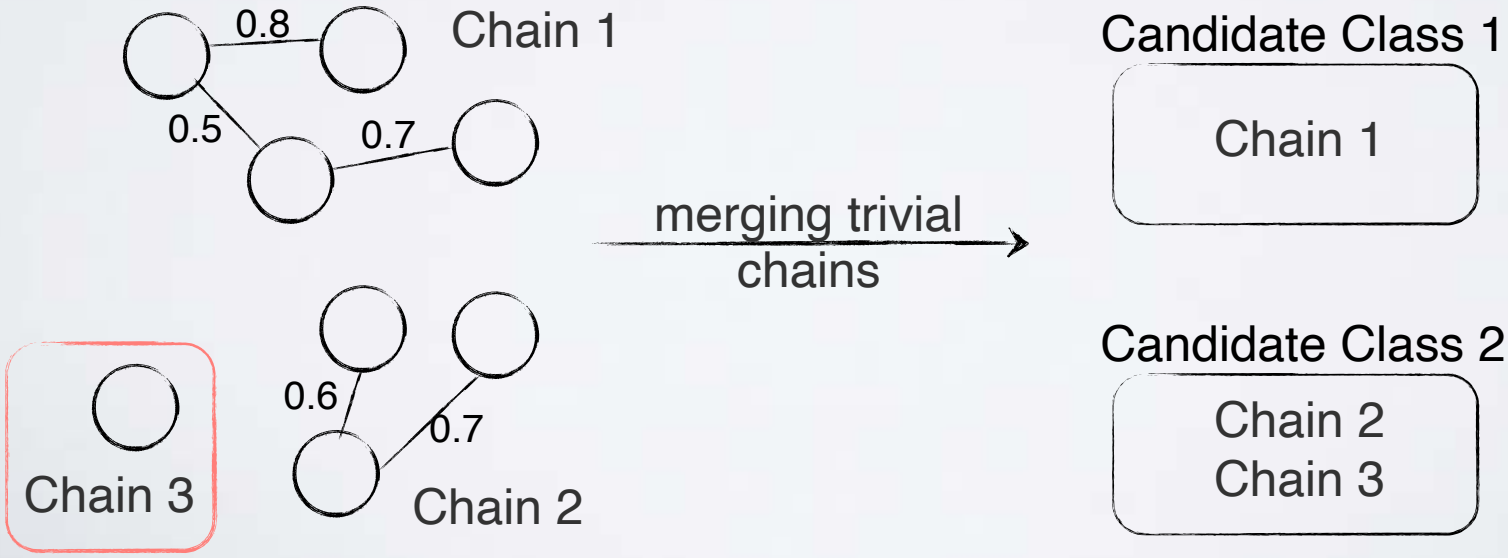
Call-based Interaction between Methods

## CSM

Conceptual Similarity between Methods

# GRAPH theory-based



# Parameters

## The weights for the similarity measures

How important are the calls between methods? the shared attributes? the semantic similarity?

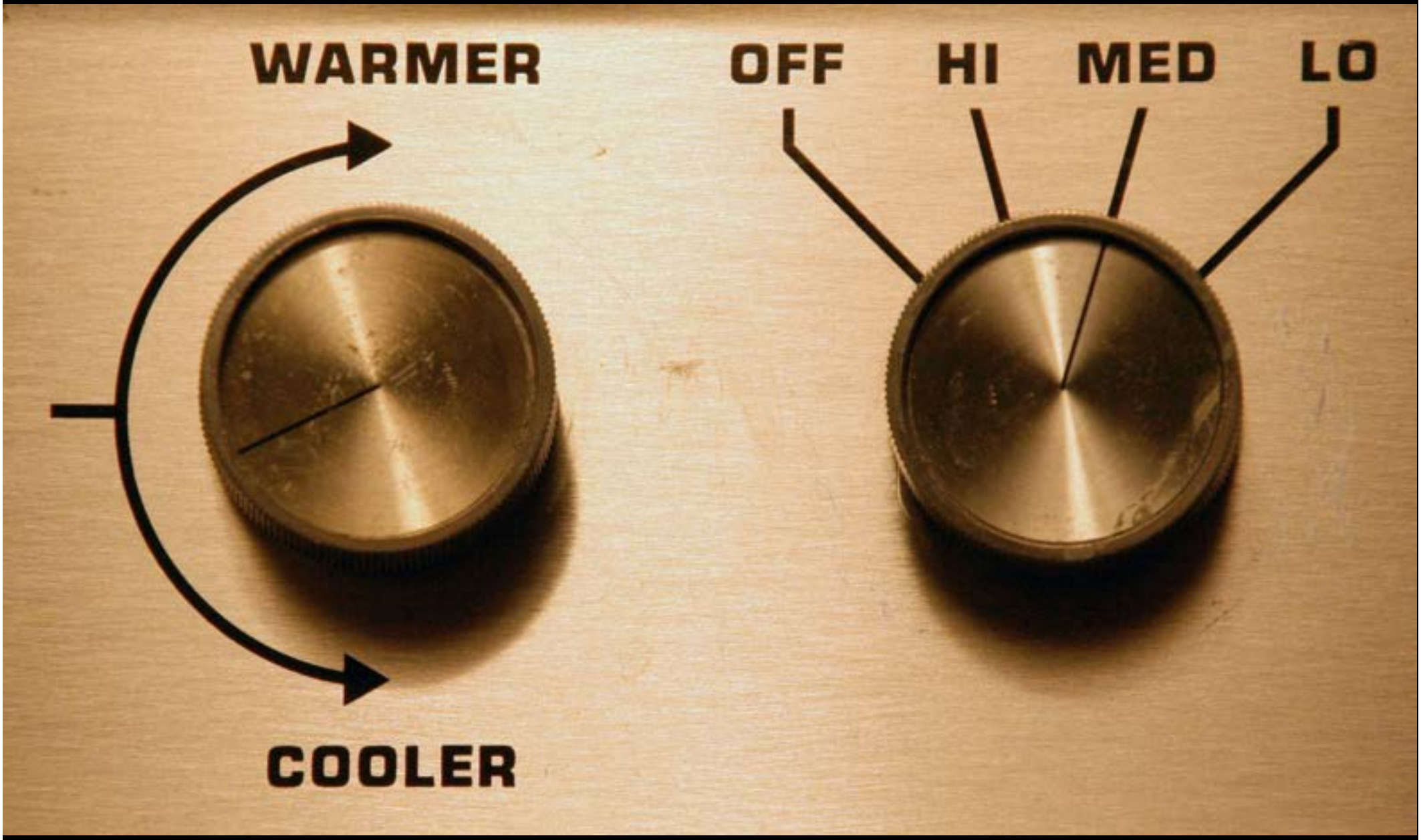
## The threshold to filter the method-by-method matrix

**Constant threshold?**

The value of the threshold is fixed a priori, e.g., 0.1

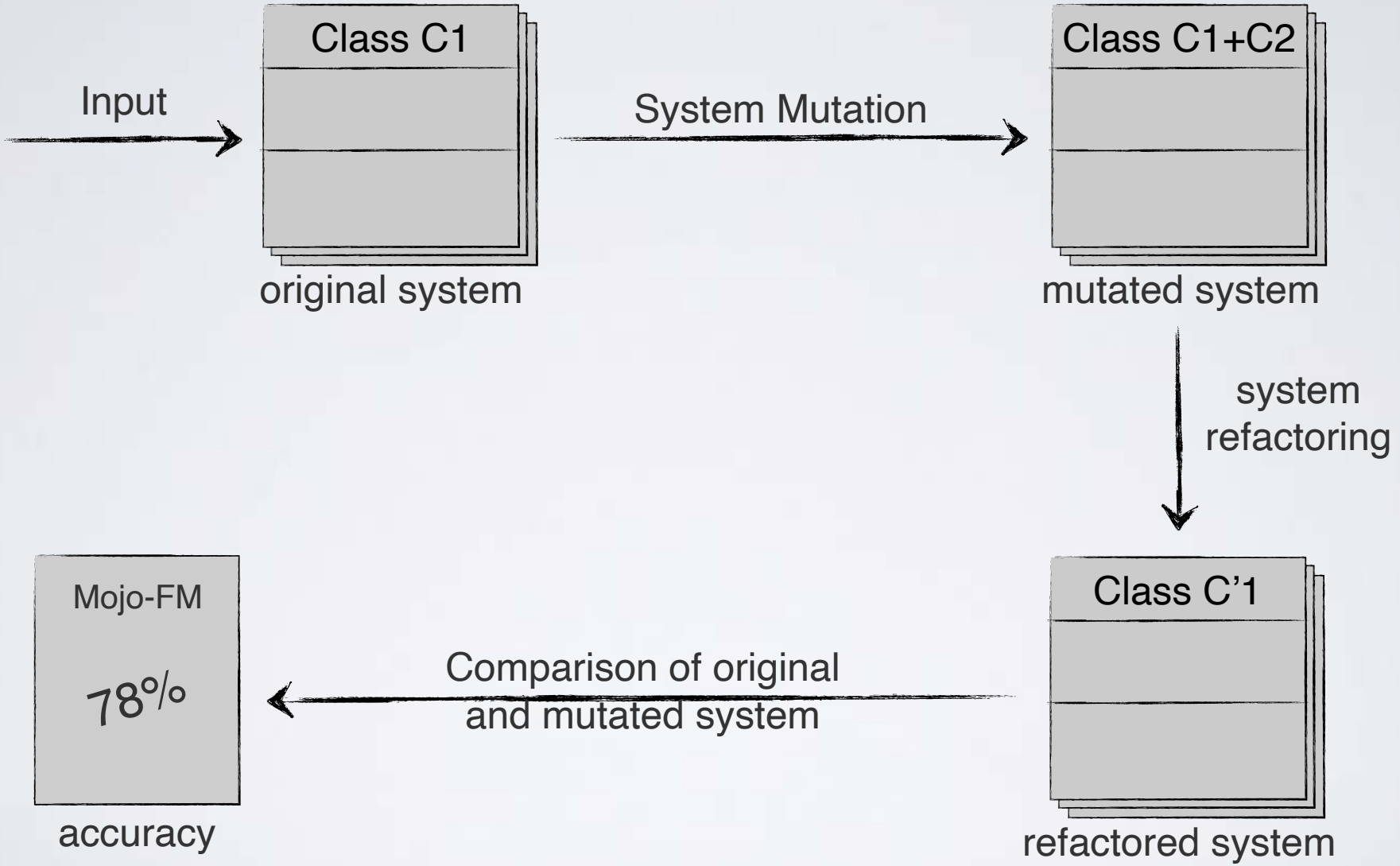
**Variable threshold?**

The value of the threshold is selected taking into account the characteristics of the provided input, e.g., the median of the values in the method-by-method matrix




# How to tune the parameters?

# A mutation testing-like approach



## Only select classes having high quality

You want your approach to extract high quality classes from a Blob.
So, merge high quality classes and consider them as oracle.

## Classes from five object systems

For each system we created 50 artificial Blobs composed by merging 2 classes and 50 composed by merging 3 classes. Total of 500 Blobs.

## Evaluated parameters

Weights for similarity measures: for each metric weight we varied this parameter starting at 0 and increasing it until 1 by a step of 0.1. All possible combinations ensuring the sum equals to 1.

Threshold to cut the method-by-method matrix: constant [0.1, 0.2, 0.3, 0.4] and variable [Q1, Q2, Q3].

A total of 462 possible combinations.

# output

## The importance of the different sources of information

Calls between methods: 0.1-0.3
Shared attributes: 0.2-0.4
Semantic similarity: 0.4-0.5

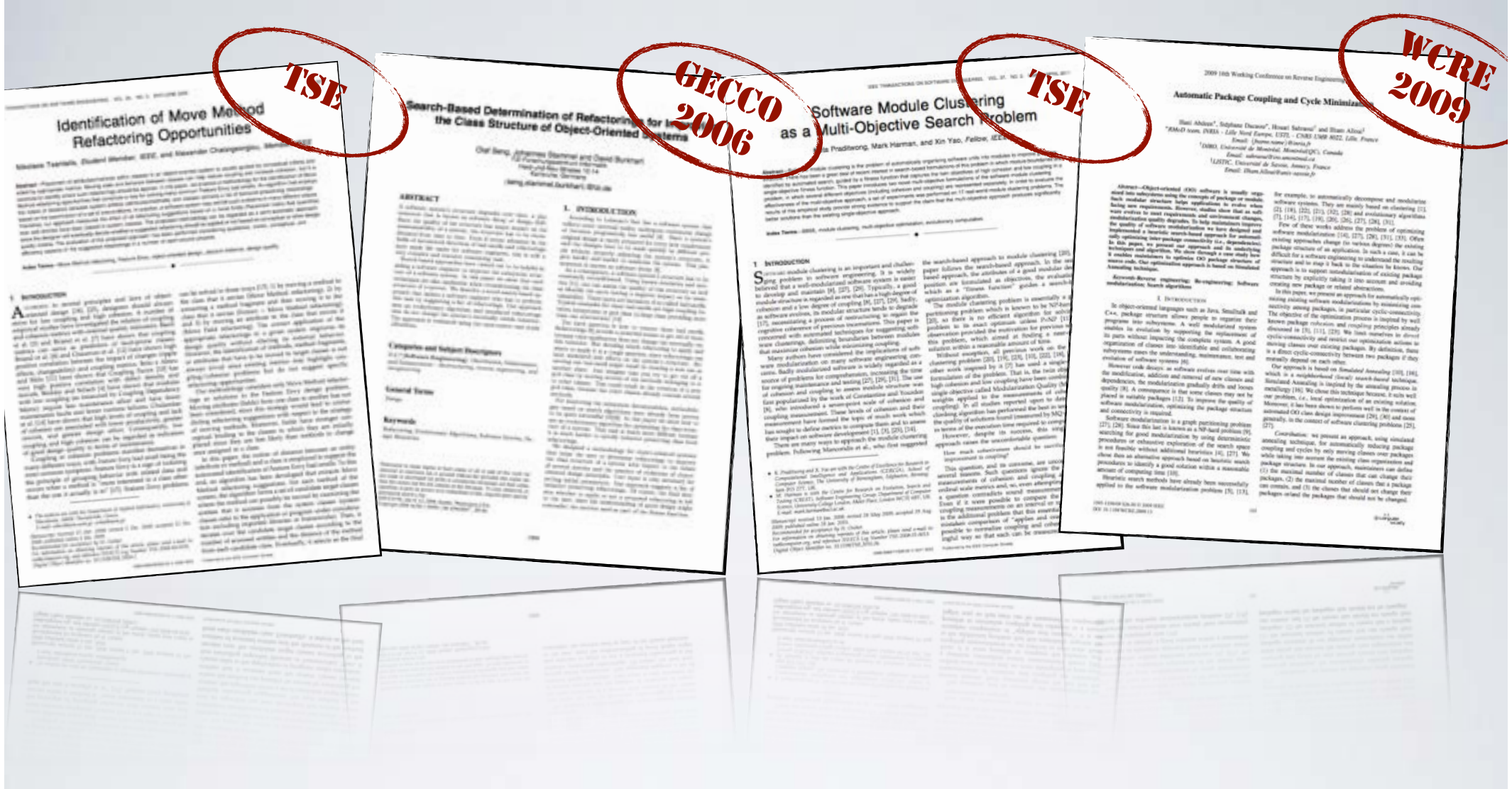
## Threshold to cut the method-by-method matrix

Median of the non-zero values present in the method-by-method matrix

## Average performances

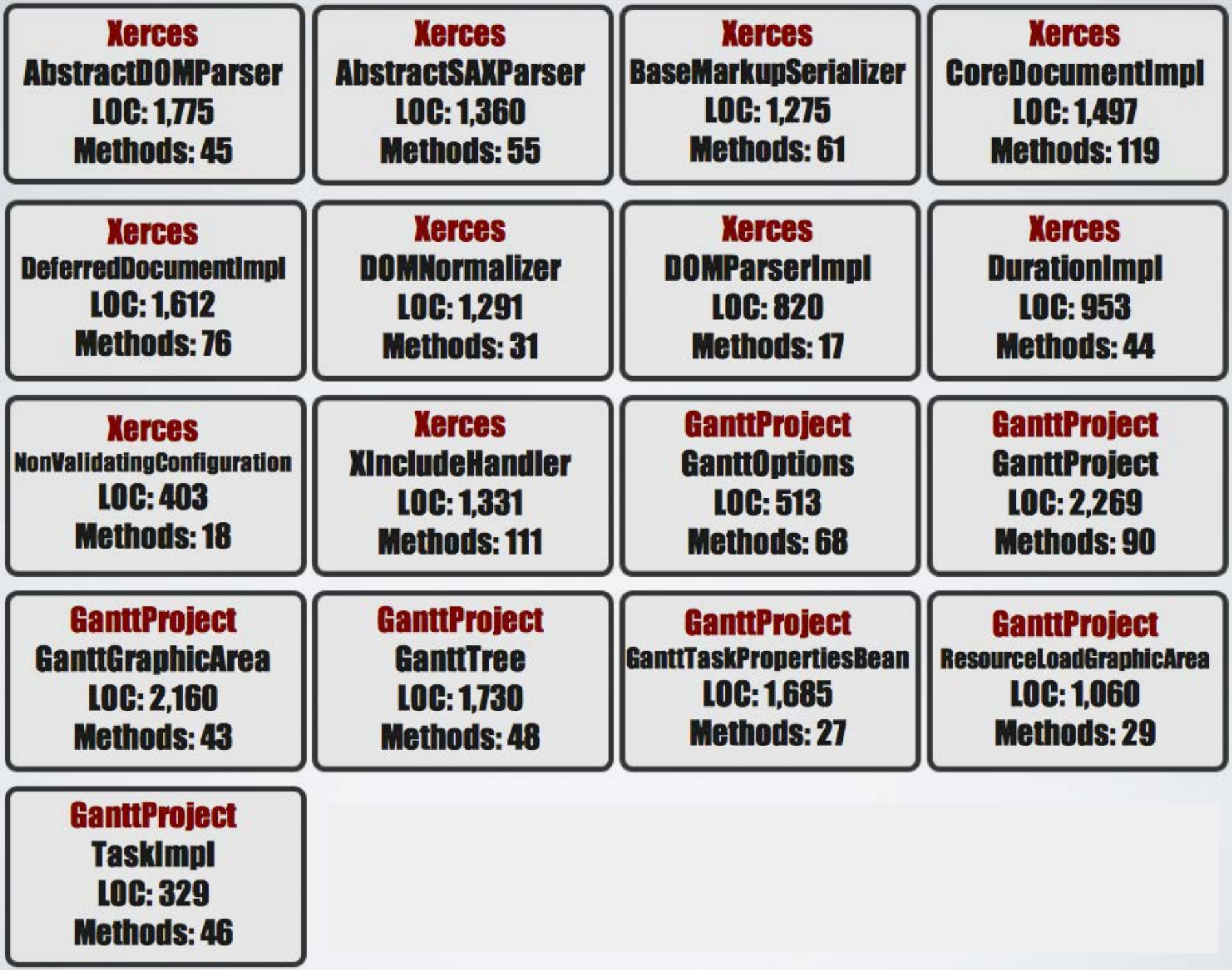
2 Merged Classes: 88%
3 Merged Classes: 75%


STOP
This is not
an evaluation

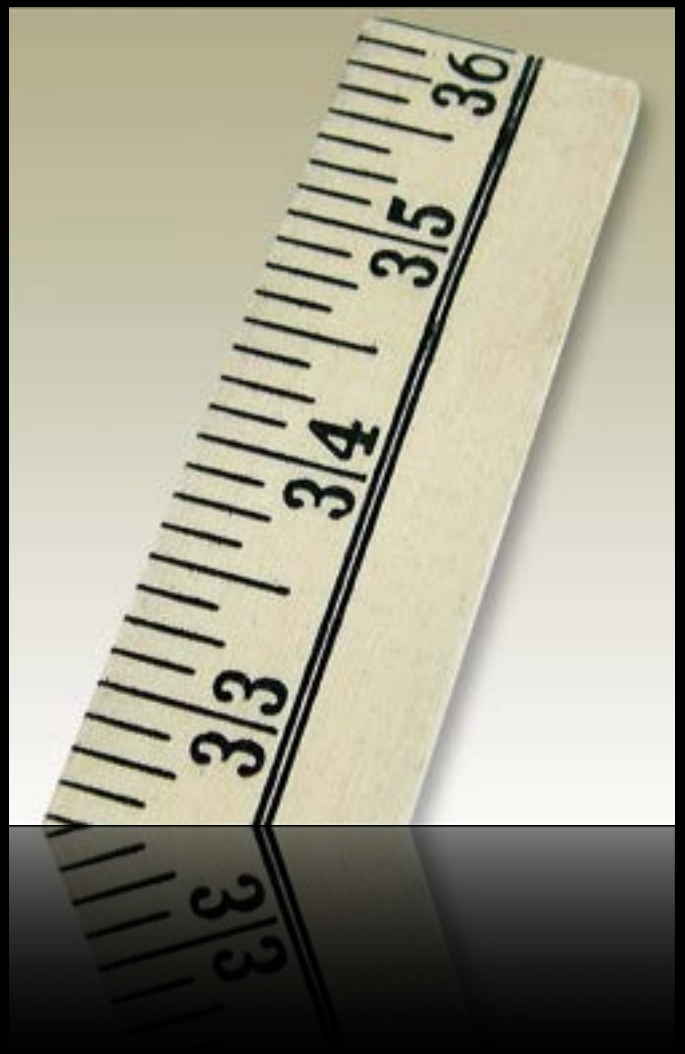



# Evaluation mainly based on software quality metrics


# Metric based evaluation


Xerces
AbstractDOMParser
LOC: 1,775
Methods: 45
Xerces
AbstractSAXParser
LOC: 1,360
Methods: 55
Xerces
BaseMarkupSerializer
LOC: 1,275
Methods: 61
Xerces
CoreDocumentImpl
LOC: 1,497
Methods: 119
Xerces
DeferredDocumentImpl
LOC: 1,612
Methods: 76
Xerces
DOMNormalizer
LOC: 1,291
Methods: 31
Xerces
DOMParserImpl
LOC: 820
Methods: 17
Xerces
DurationImpl
LOC: 953
Methods: 44
Xerces
NonValidatingConfiguration
LOC: 403
Methods: 18
Xerces
XIncludeHandler
LOC: 1,331
Methods: 111
GanttProject
GanttOptions
LOC: 513
Methods: 68
GanttProject
GanttProject
LOC: 2,269
Methods: 90
GanttProject
GanttGraphicArea
LOC: 2,160
Methods: 43
GanttProject
GanttTree
LOC: 1,730
Methods: 48
GanttProject
GanttTaskPropertiesBean
LOC: 1,685
Methods: 27
GanttProject
ResourceLoadGraphicArea
LOC: 1,060
Methods: 29
GanttProject
TaskImpl
LOC: 329
Methods: 46

# Considered Metrics



**Lack of Cohesion of Methods (LCOM)**

Pairs of methods in a class not sharing any attribute

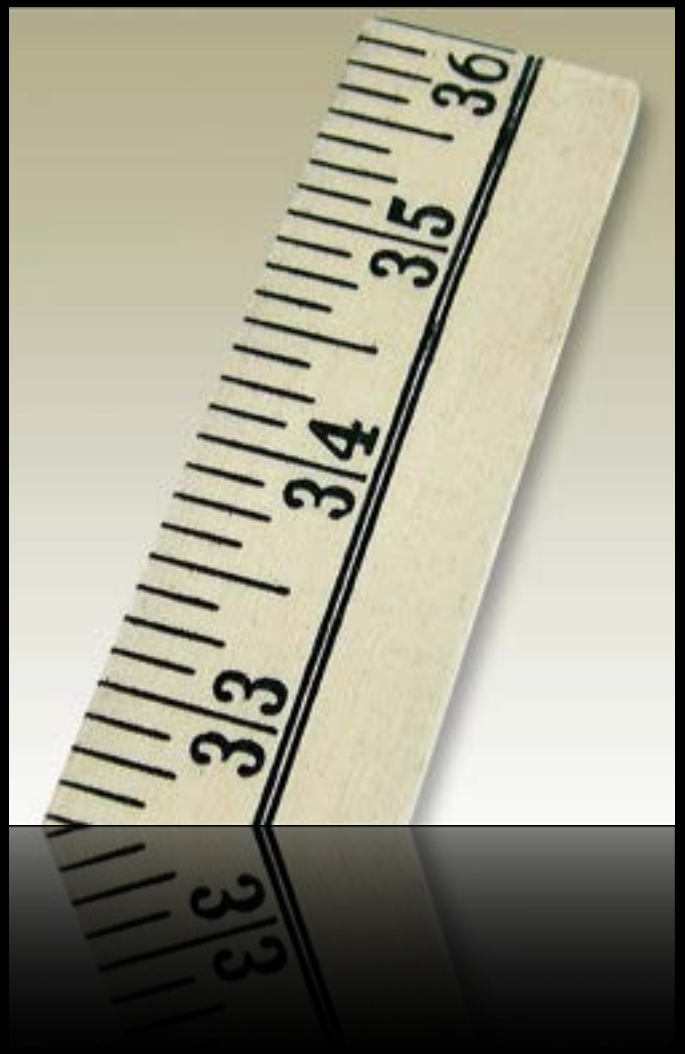
**Conceptual Cohesion of Classes (C3)**

Semantic similarity between methods of a class

**Message Passing Coupling (MPC)**

The number of calls between a pair of classes

# Considered Metrics



**Lack of Cohesion of Methods (LCOM)**

+82%

**Conceptual Cohesion of Classes (C3)**

+105%

**Message Passing Coupling (MPC)**

+3%

# Can we draw any conclusion?

The problem with this paper, as with all other that I have read, lies in the improper validation.

The first experiment tests the quality of the proposed approach using metrics that were never guardanteed to indicate a good design.

Design decisions are more intricate and delicate than just trying to minimize coupling and maximize cohesion.

I also agree with the threat to validity identified by the authors, that the evaluation metrics are very similar to the exploited metrics with which is a fundamental problem that cannot just be acknowledged.

For these reasons, I would totally reject the results of the first experiment.

# Can we draw any conclusion?

Quality metrics can help you in performing a first assessment of the approach

A refactoring technique should not increase coupling and/or reduce cohesion

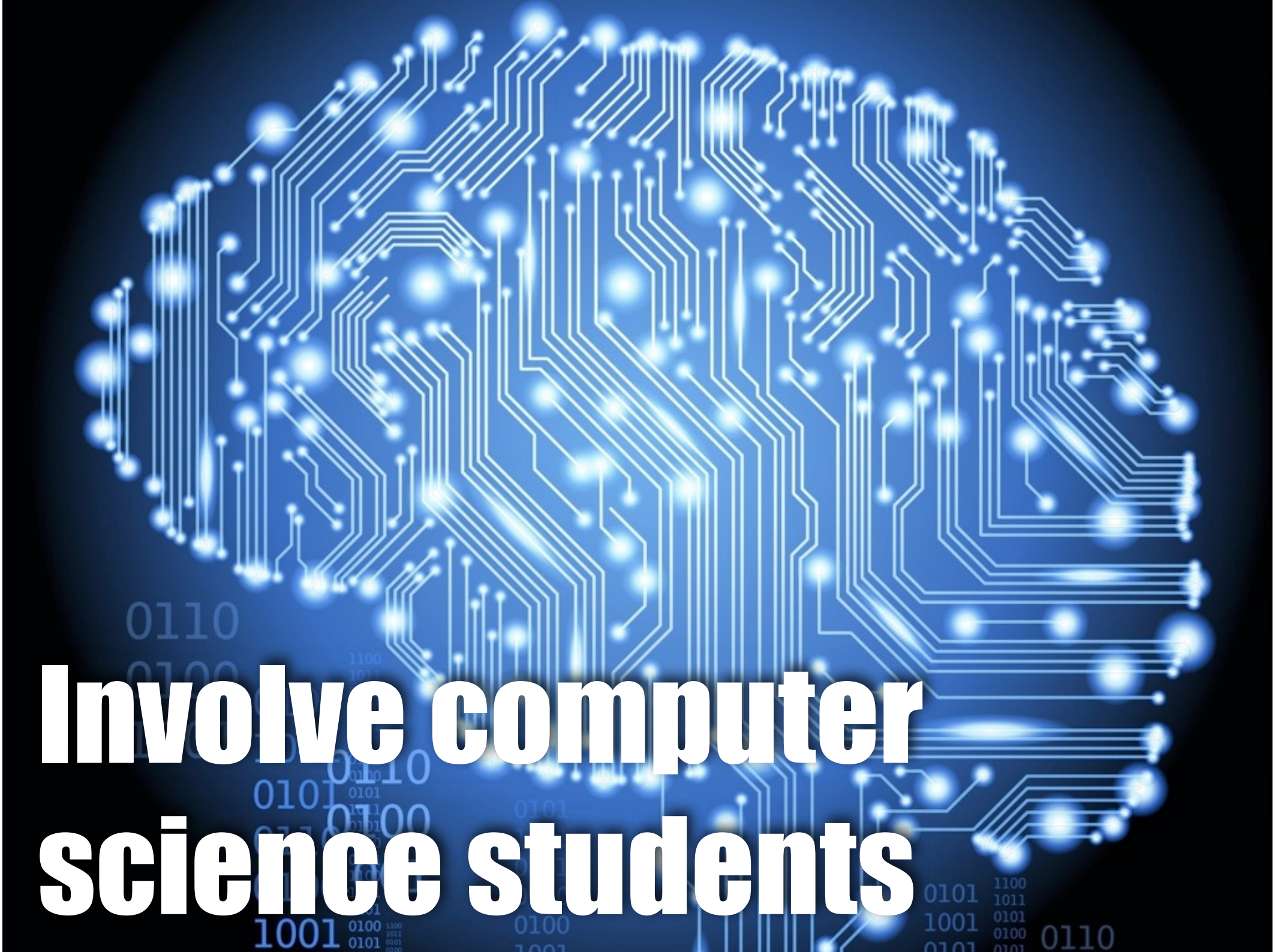
A quality metric-based evaluation cannot be the core of your evaluation

[...] I particularly liked Section 4

# Guess what's inside Section 4?

# Involve Software Developers


Involving Developers
is difficult


Involve computer
science students

# Meaningfulness of the suggested refactoring operations

## Subjects

50 Master's students

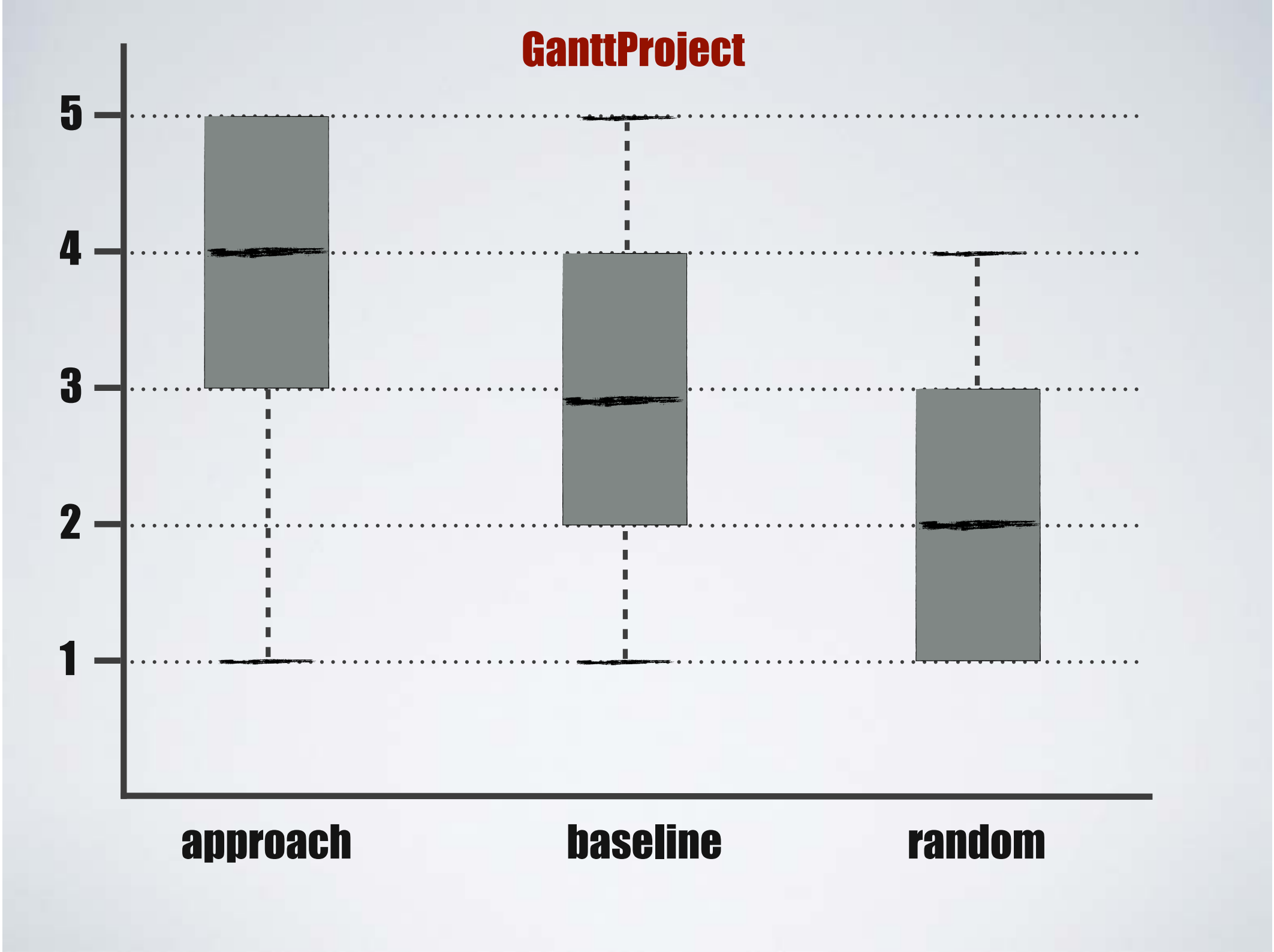
## Objects

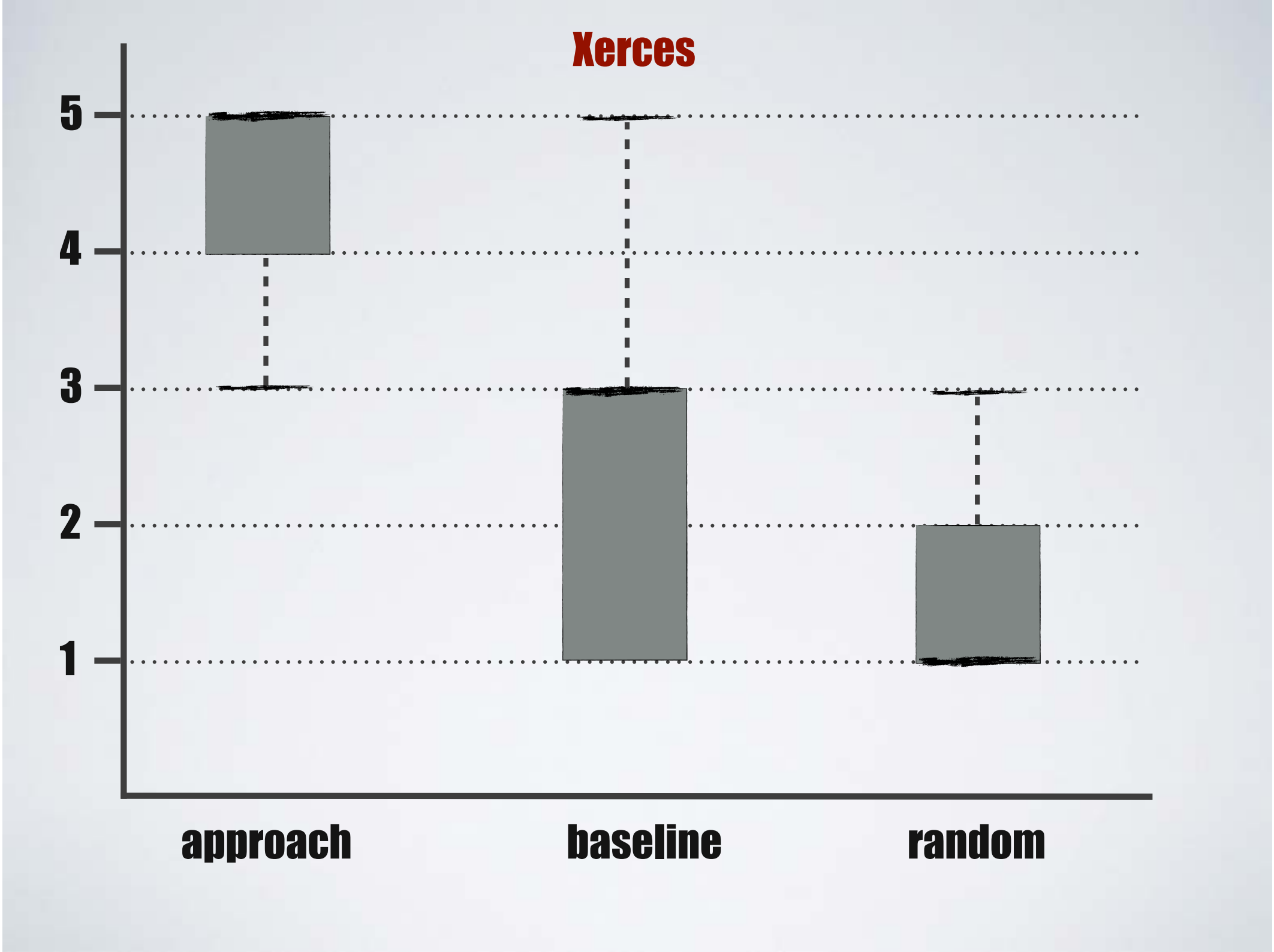
17 Blobs used in the metric-based evaluation

## What we asked

**Agreement to the claim:** "The proposed refactoring results in a better division of responsibilities"

**Five point Likert scale:** "1: Strongly disagree; 2: Disagree; 3: Neutral; 4: Agree; 5: Fully agree"


GanttProject
5
4
3
2
1
approach
baseline
random


Xerces
5
4
3
2
1
approach
baseline
random

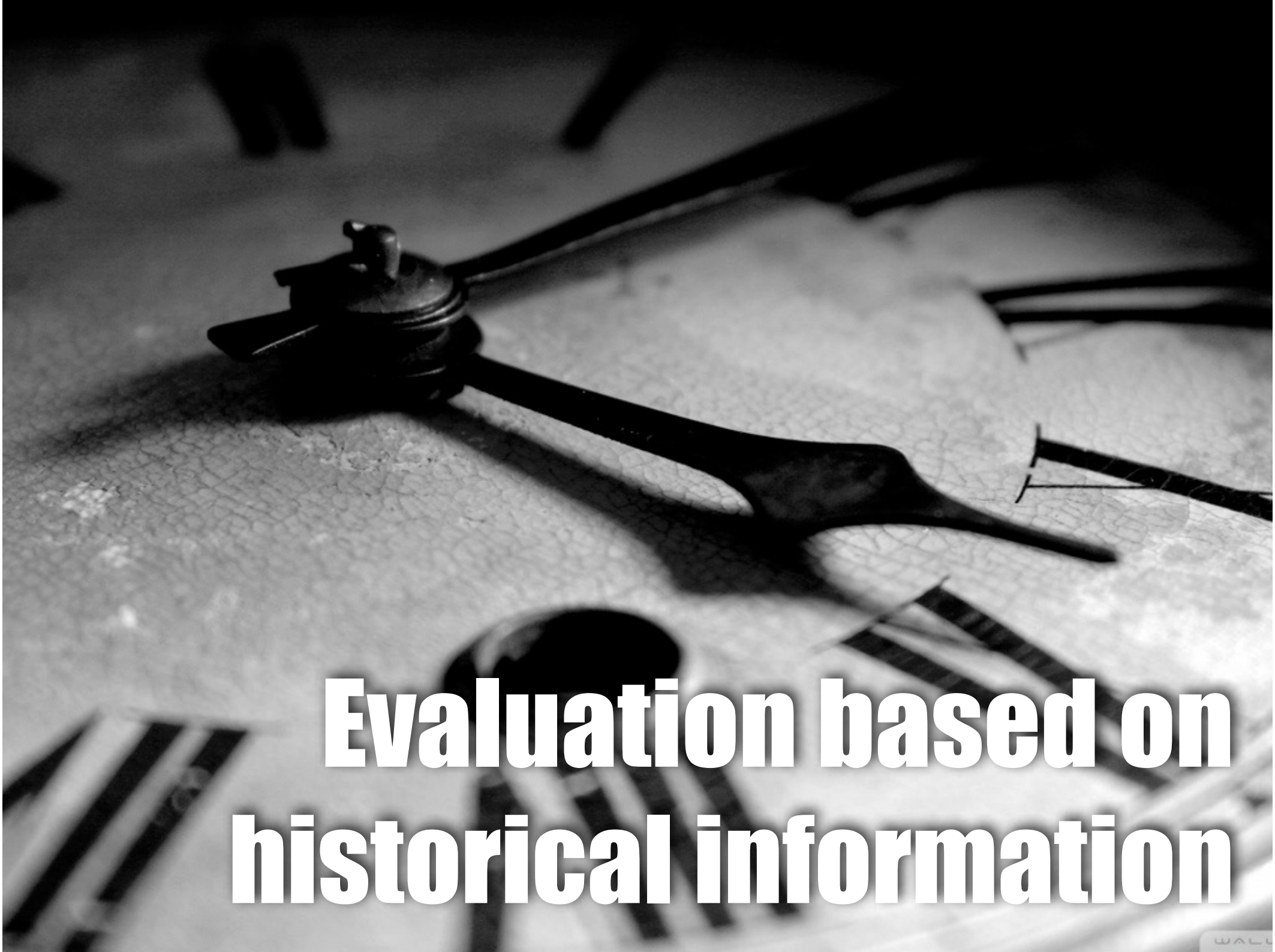
The question asked to the students "are the responsibilities better separated" is not sufficient.

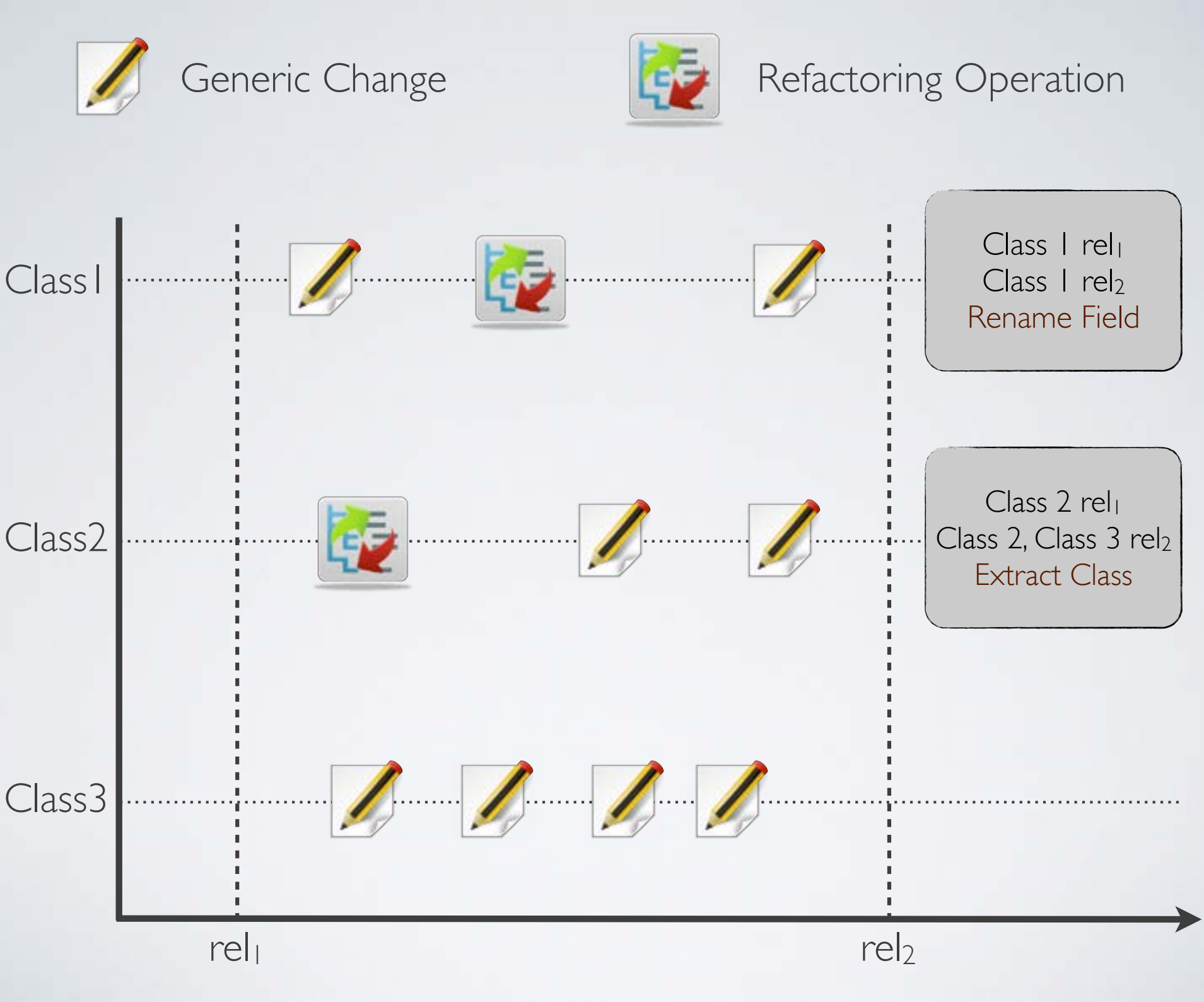
This is not the right question to ask, IMHO. The right questions are "would you split the class differently? how? why?" and "would you have split the class at all? why?". In other words, I see no hypothetical/theoretical reason how your experiment could have invalidated your method. It is logically and psychologically improbable the students would answer anything else but "yes, responsibilities are separated better".

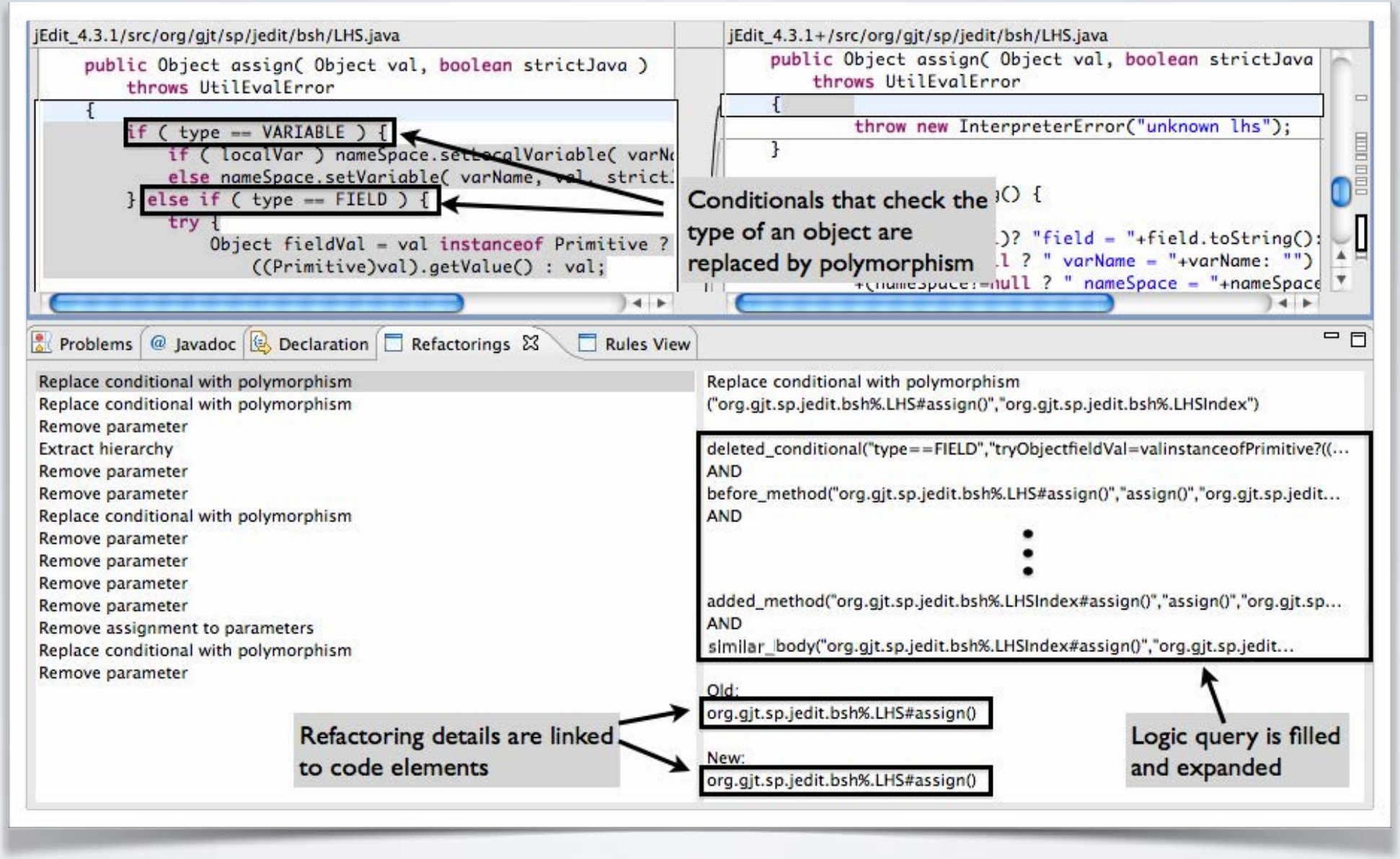
What else did they think? What else did they say?


QUANTITY


# Missing qualitative data


Evaluation based on
historical information


Generic Change
Refactoring Operation
Class1
Class2
Class3
rel1
rel2
Class 1 rel1
Class 1 rel2
Rename Field
Class 2 rel1
Class 2, Class 3 rel2
Extract Class



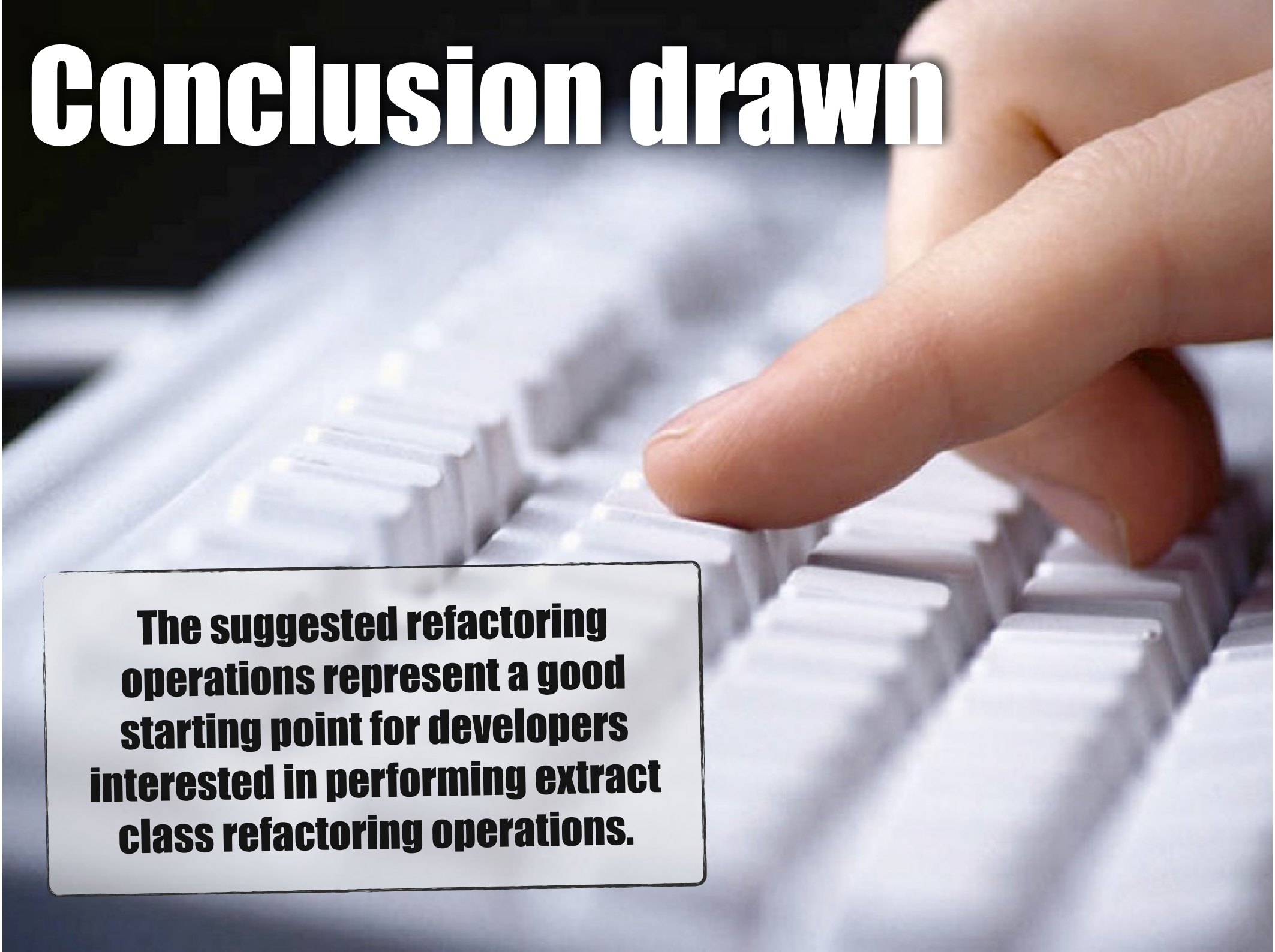

K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim,
*Template-based reconstruction of complex refactorings,*
Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010)
Download: https://webspace.utexas.edu/kp9746/www/reffinder/

| System | Original Class | Extracted Classes |
|---|---|---|
| Apache HSQLDB | Database (41) | Database (27) |
| | | SchemaManager(14) |
| | Select (14) | Select (7) |
| | | Result (7) |
| | UserManager (13) | UserManager (10) |
| | | GranteeManager (3) |
| ArgoUML | FileGeneratorAdapter (9) | FileGeneratorAdapter (3) |
| | | TempFileUtils (6) |
| | Import (10) | Import (7) |
| | | ImportCommon (3) |
| JEdit | JEditTextArea (214) | JEditTextArea (22) |
| | | SelectionManager (11) |
| | | TextArea (181) |
| JFreeChart | JFreeChart (24) | JFreeChart (16) |
| | | Plot (8) |
| | NumberAxis (20) | NumberAxis (16) |
| | | ValueAxis (4) |
| JHotDraw | DefaultApplicationModel (14) | DefaultApplicationModel (4) |
| | | AbstractApplicationModel (10) |
| Xerces | XMLDTDValidator (69) | XMLDTDValidator (38) |
| | | XMLDTDProcessor (31) |
| | XMLSerializer (25) | XMLSerializer (12) |
| | | DOMWriterImpl (13) |

# How far is the refactoring proposed by our tool from the refactoring of original developers ?

| Class | MoJoFM |
|---|---|
| Database (41) | 97% |
| Select (14) | 83% |
| UserManager (13) | 93% |
| FileGeneratorAdapter (9) | 86% |
| Import (10) | 100% |
| JEditTextArea (214) | 84% |
| JFreeChart (24) | 95% |
| NumberAxis (20) | 94% |
| DefaultApplicationModel (14) | 92% |
| XMLDTDValidator (69) | 88% |
| XMLSerializer (25) | 91% |
| **Average** | **91%** |

# Conclusion drawn



The suggested refactoring operations represent a good starting point for developers interested in performing extract class refactoring operations.

Evaluation based on historical information looks like a good idea. However, we cannot know the reasons behind the refactoring performed by developers nor if they have been performed manually or by using automatic tools.

[...]

**We need to involve developers in the evaluation of the proposed refactoring ...**

**... but in a different way than in the previous user study**

## Subjects

15 Master's students

## Objects

11 Blobs refactored by original developers

## What we asked

Would you split this class?

- if YES:
  - i. Why?
  - ii. Would you split the class differently than the provided refactoring solution? Why?
  - iii. Did you find the provided refactoring solution useful as starting point to perform your refactoring?Why?
- if NO:
  - i. Why?:

| Class | % Students answering YES | | |
|---|---|---|---|
| | Would you split this class? | Would you split the class differently? | Was the provided refactoring suggestion useful? |
| Database | 87% | 62% | 100% |
| Select | 27% | 0% | 100% |
| UserManager | 100% | 87% | 67% |
| FileGeneratorAdapter | 67% | 40% | 100% |
| Import | 40% | 0% | 100% |
| JEditTextArea | 100% | 100% | 100% |
| JFreeChart | 100% | 0% | 100% |
| NumberAxis | 87% | 62% | 100% |
| DefaultApplicationModel | 80% | 8% | 100% |
| XMLDTDValidator | 100% | 0% | 100% |
| XMLSerializer | 87% | 67% | 100% |

# Usefulness of the provided refactoring

**1. it eases code comprehension;**

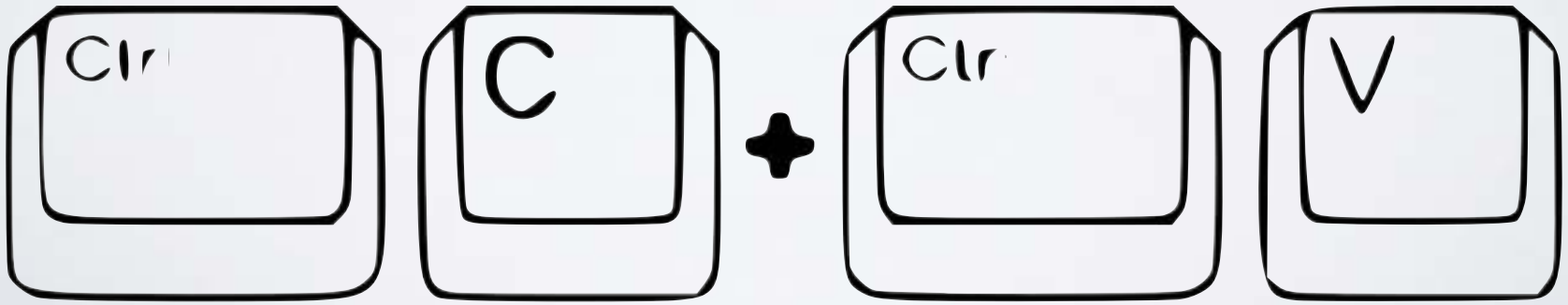
**2. it highlights the main responsibilities implemented in a class;**

**3. the extracted classes are cohesive.**

*"without the refactoring suggestion it would be too difficult to identify the main responsibilities of the classes"*

# MoJoFM between refactorings performed by students and those by the original developers

# 93%



# The suggested refactorings are meaningful and useful for developers.

Da: Andrian Marcus
Oggetto: **Re: Fwd: Decision on your Manuscript #EMSE-D-12-00704R1**
Data: 26 febbraio 2013 14.06.51 GMT+01.00
A: Andrea De Lucia
Cc: Rocco Oliveto <rocco.oliveto@gmail.com>, Bavota Gabriele

Finally!!!

This paper has a history. You can almost write a paper about this paper :)

# Move Class Refactoring

G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, A. De Lucia.
*Improving Software Modularization via Automated Analysis of Latent Topics and Dependencies.*
Transactions on Software Engineering and Methodology (TOSEM), 2014

# The design problem

> One of the main reasons for architectural erosion is inconsistent placement of source code classes in software packages
> [Fowler 1999]

Fowler M.
*Refactoring: improving the design of existing code*
**Addison-Wesley**

# Big-bang re-modularization

Software Module Clustering as a Multi-Objective Search Problem

Kata Praditwong, Mark Harman, and Xin Yao, Fellow, IEEE

Automatic Package Coupling and Cycle Minimization

Experiments with Clustering as a Software Remodularization Method*

Nicolas Anquetil and Timothy C. Lethbridge

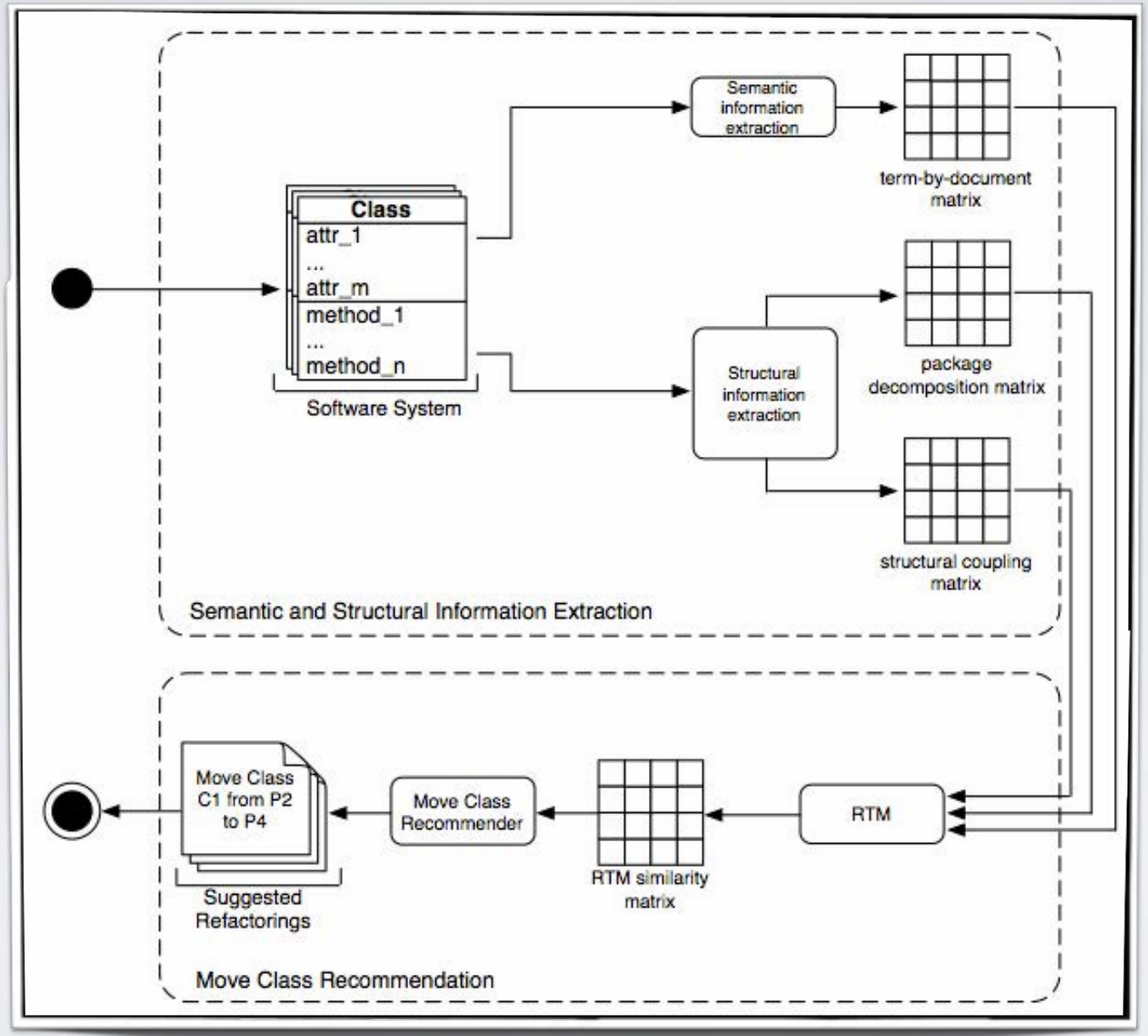
# R3: Rational Refactoring via RTM

R3 performs a fine-grained re-modularization through move class refactorings

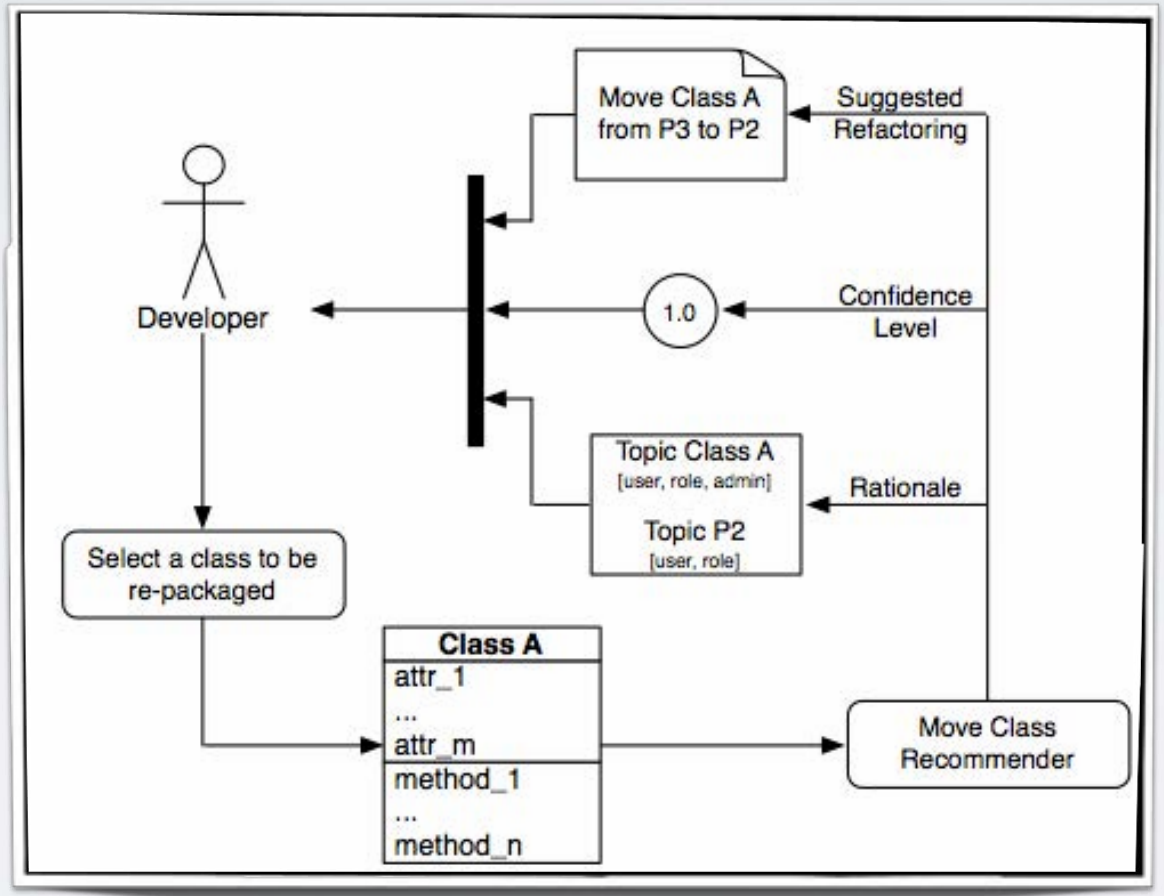
R3 takes into account the **developers'** original **system decomposition** (besides other information)

R3 provides explanations to the **developer** about the performed refactoring operations

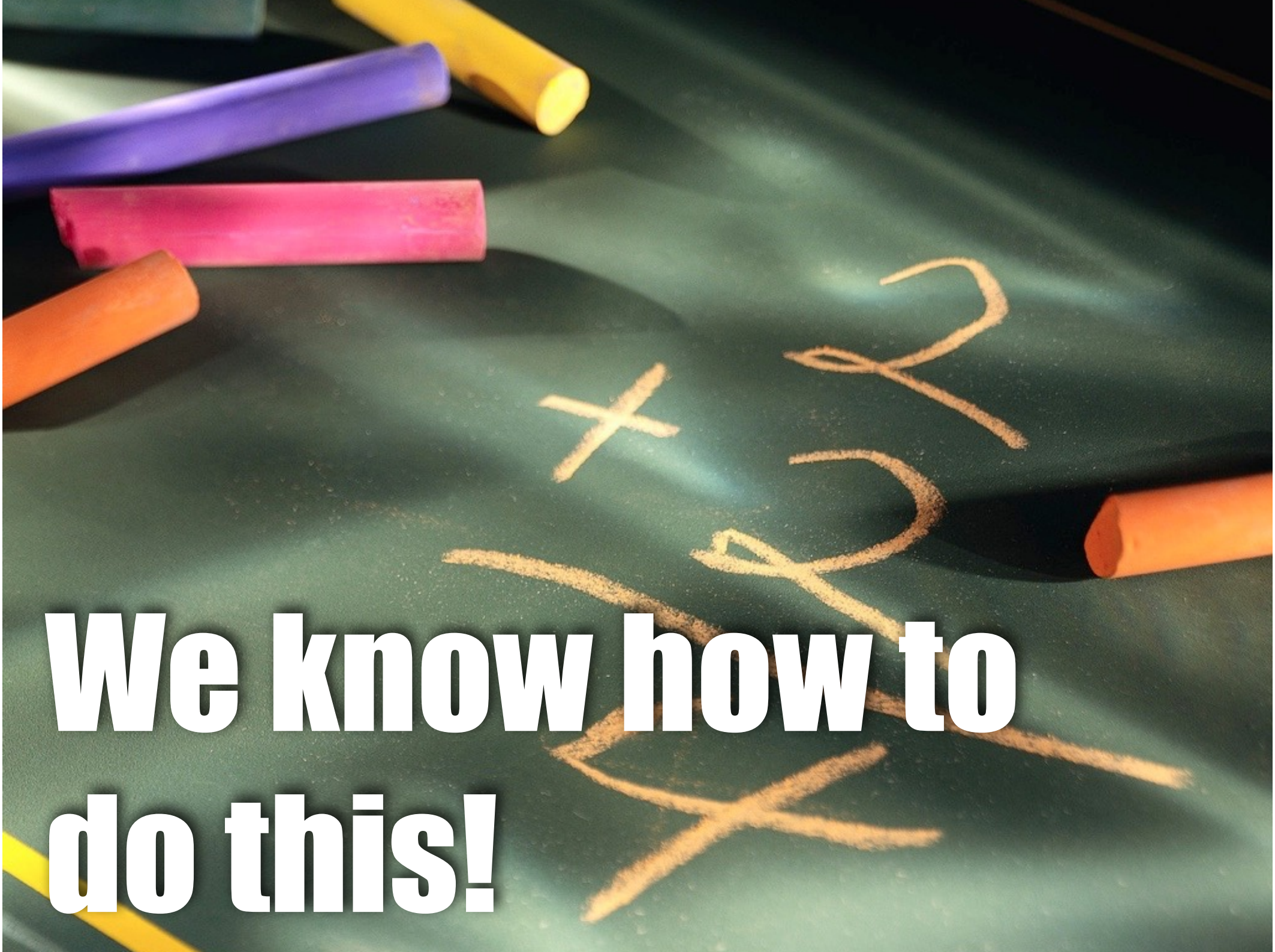
# R3: the process



# R3: feedbacks provided to developers




VERY SATIS
6
7
8
9
Evaluation


We know how to
do this!

# Quality Metrics

## 9 Systems

**High Confidence Level**

Coupling up to

**-86%**

Cohesion up to

**140%**

**Low Confidence Level**

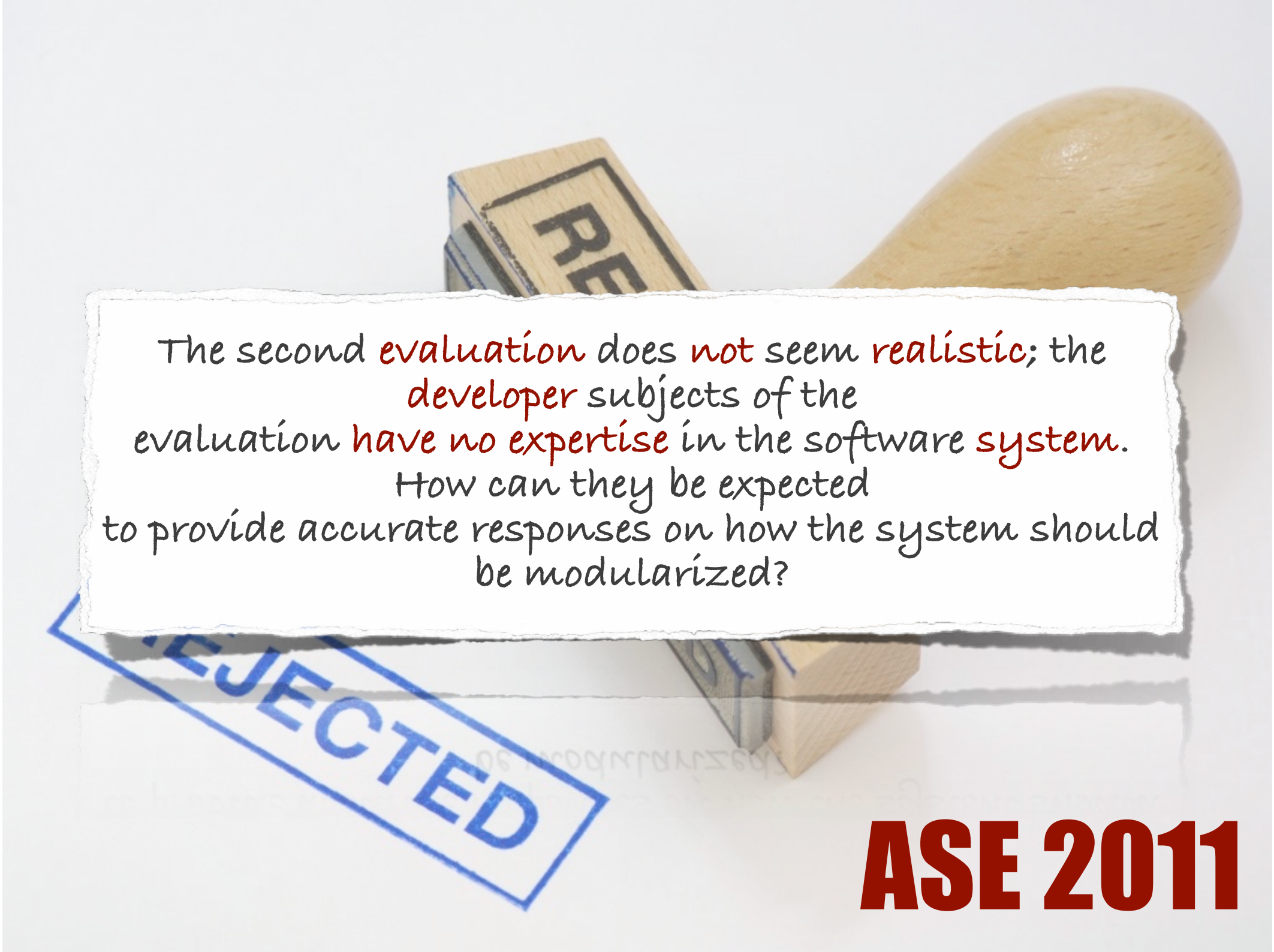
Not Reliable Suggestions

# 48 External Developers:

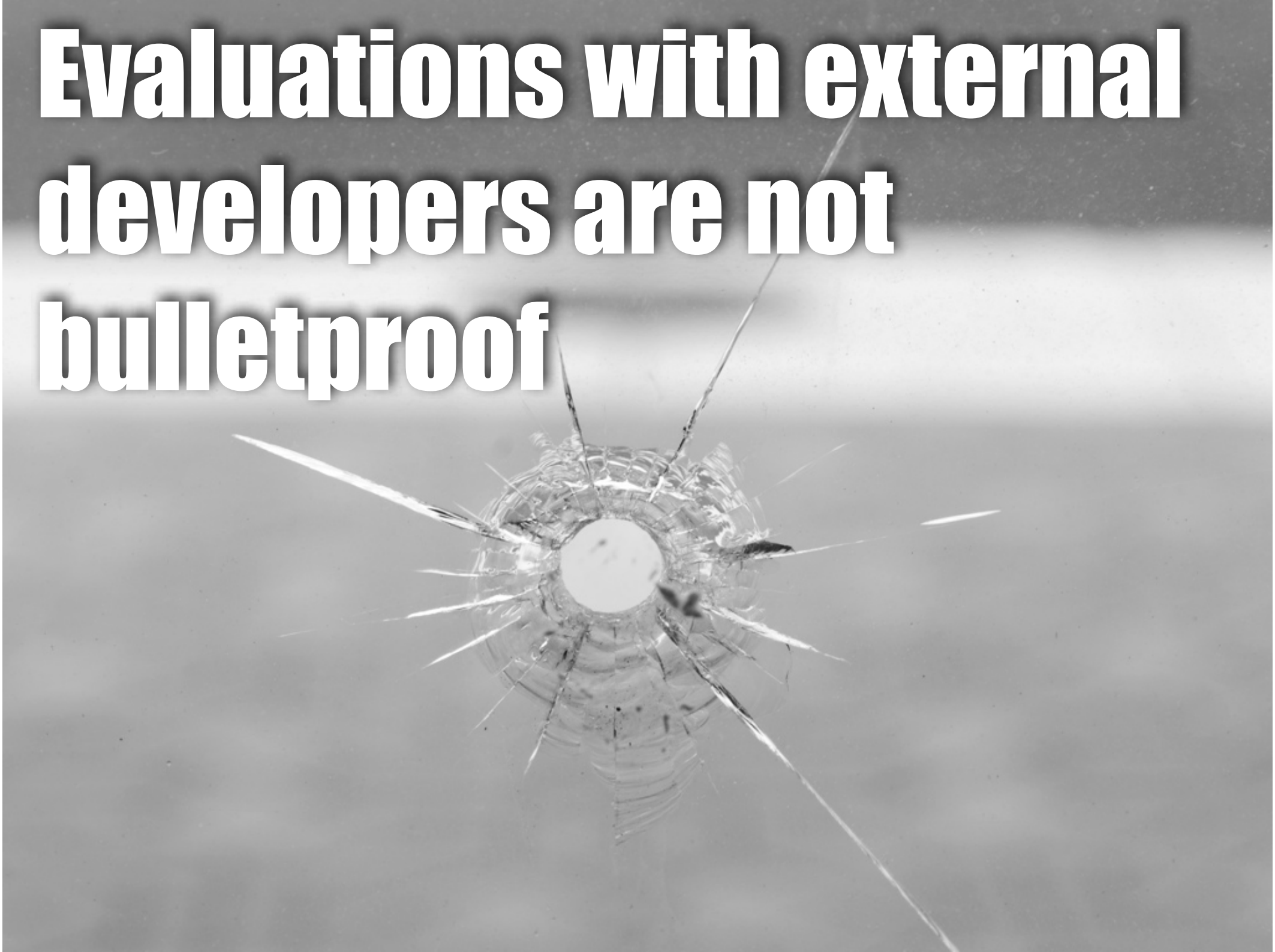
**evaluated the meaningfulness of the refactorings and explanations proposed by R3**

| | |
|---|---|
| Meaningful suggestions in the high confidence scenario | 75% |
| Meaningful explanations in the high confidence scenario | 55% |


REJECTED
REJECTED
ASE 2011


The second evaluation does not seem realistic; the developer subjects of the evaluation have no expertise in the software system. How can they be expected to provide accurate responses on how the system should be modularized?
REJECTED
ASE 2011

# Evaluations with external developers are not bulletproof



# Quality Metrics

## 9 Systems

**High Confidence Level**

Coupling up to

**-86%**

Cohesion up to

**140%**

**Low Confidence Level**

Not Reliable Suggestions

## 48 External Developers:

evaluated the meaningfulness of the refactorings and explanations proposed by R3

| | |
|---|---|
| Meaningful suggestions in the high confidence scenario | 75% |
| Meaningful explanations in the high confidence scenario | 55% |

## 14 Original Developers - 4 systems:

answered to the question: "would you apply the proposed refactoring?" and evaluated the provided rational

| | |
|---|---|
| Meaningful suggestions in the high confidence scenario | 80% |
| Meaningful explanations in the high confidence scenario | 70% |

# Do we need both original and external developers?

# Original

Deep knowledge of the system

Rational behind all design decisions

Could be authors of bad design choices

# External

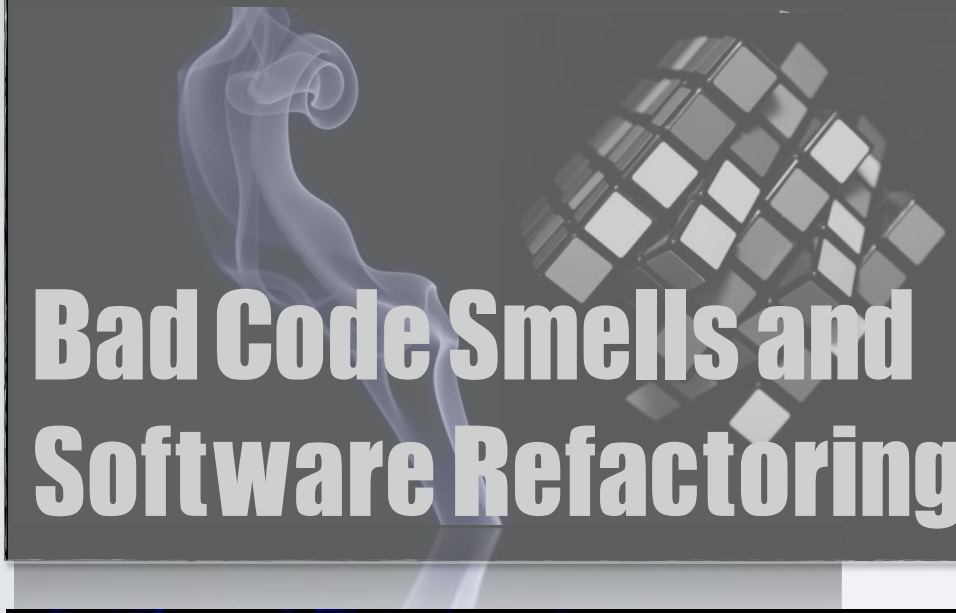
Poor knowledge of the system

No rational behind design decisions

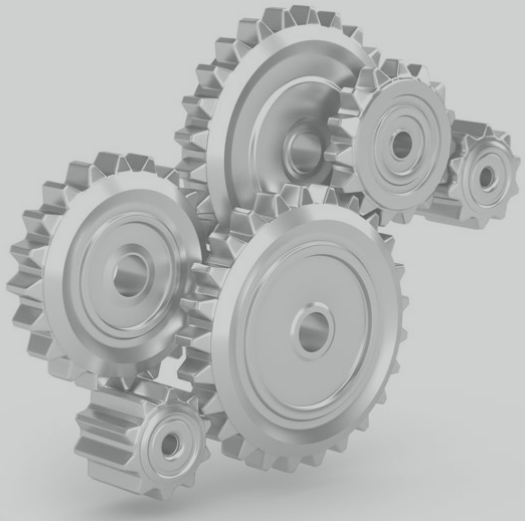
Objective evaluation of the refactorings

# Part I

## Bad Code Smells and Software Refactoring

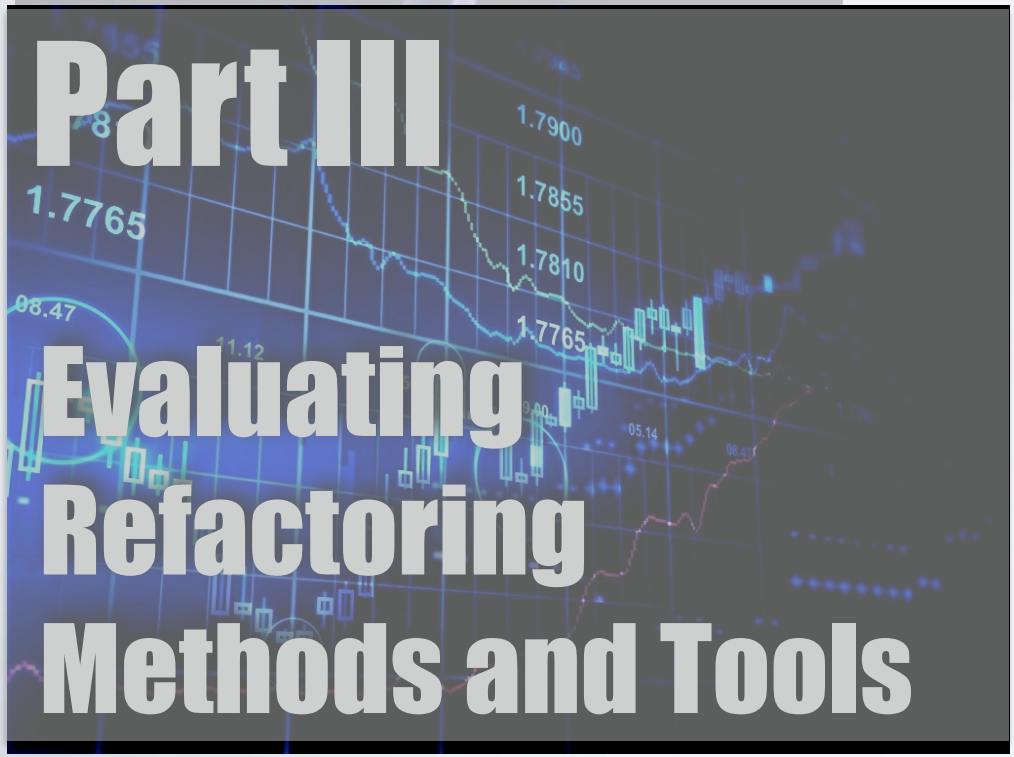


# Part II



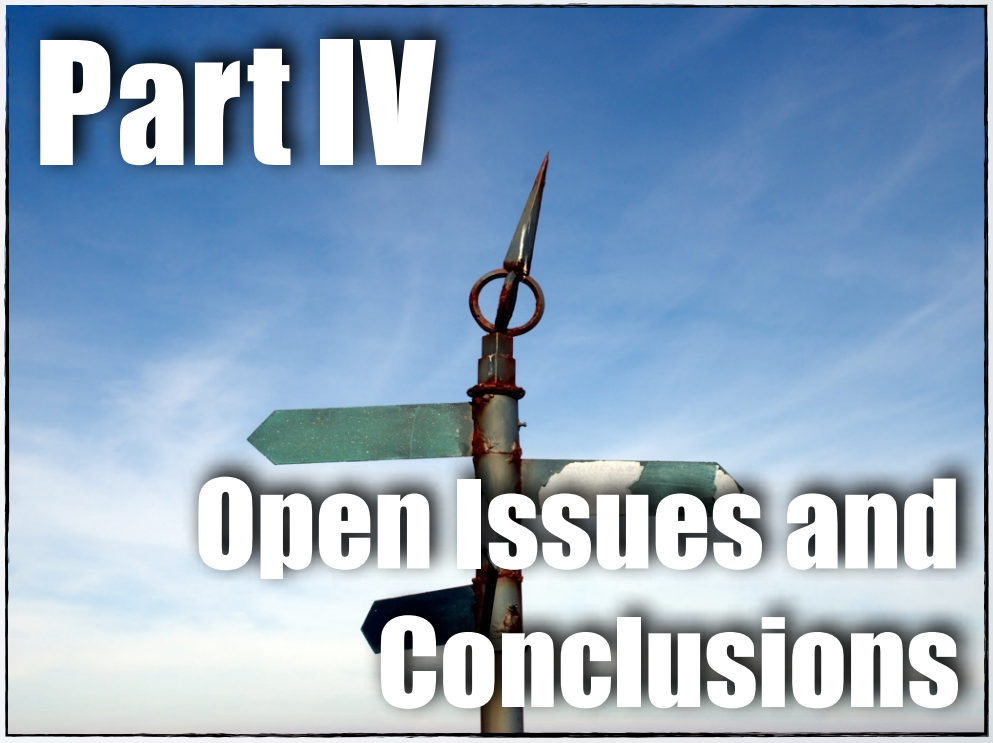
## The refactoring process

# Part III



## Evaluating Refactoring Methods and Tools

# Part IV

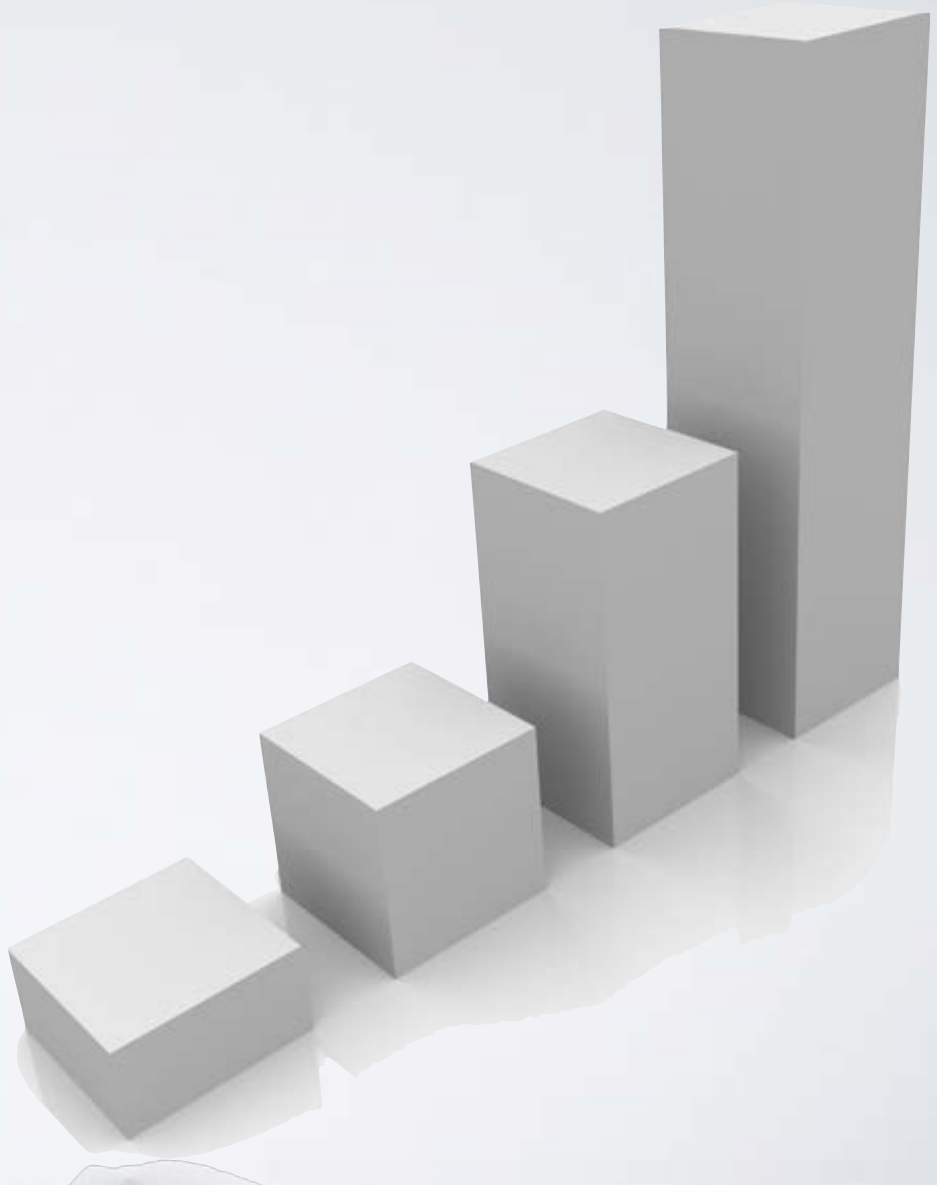


## Open Issues and Conclusions

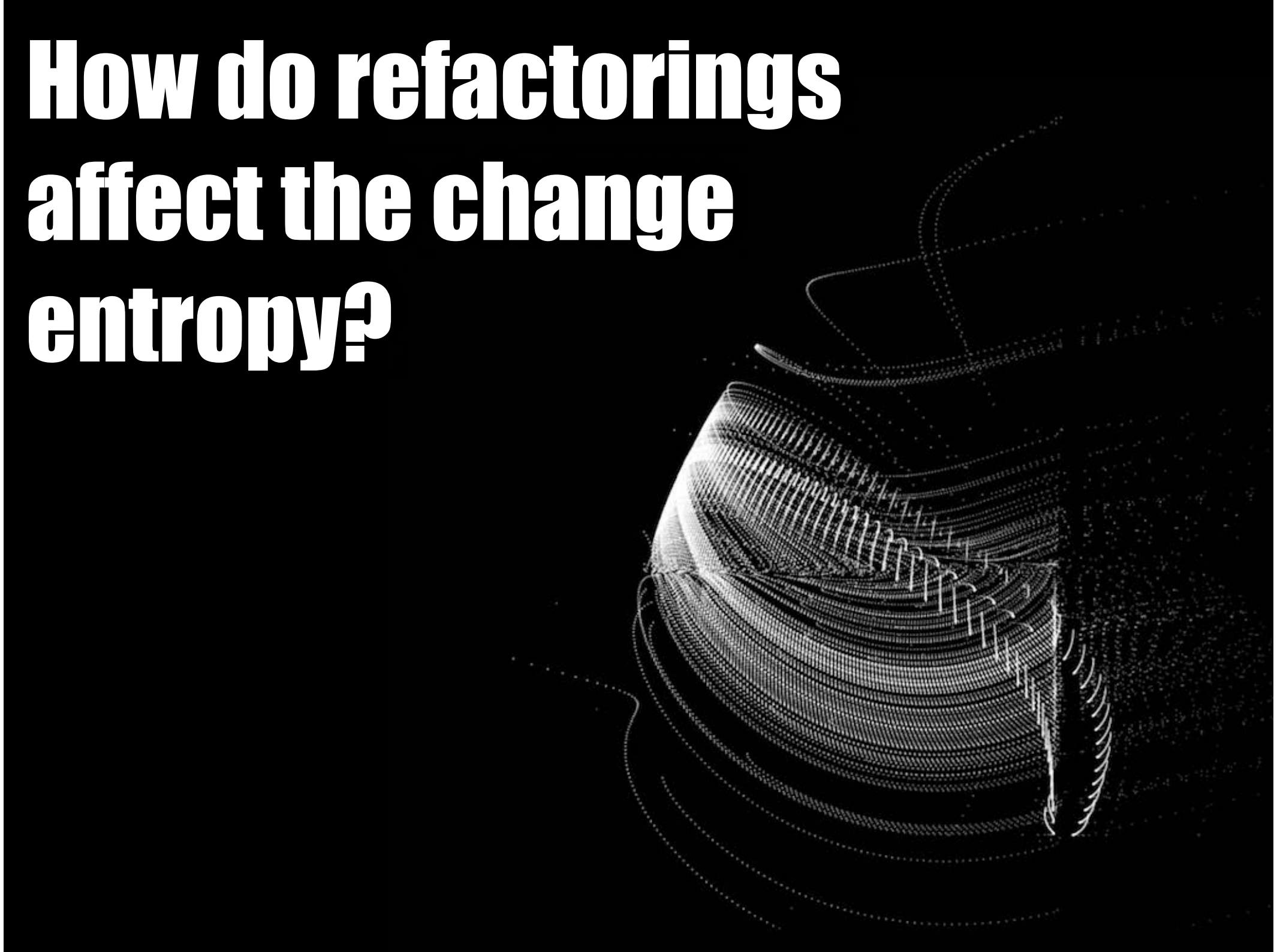
# Benefits and Side effects

# Refactoring benefits

complexity
extensibility
modularity
reusability
maintainability



# How do refactorings affect the change entropy?



Empir Software Eng (2014) 19:1–38
DOI 10.1007/s10664-012-9214-z

# How changes affect software entropy: an empirical study

**Gerardo Canfora · Luigi Cerulo · Marta Cimitile · Massimiliano Di Penta**

Published online: 14 July 2012

**Editor:** Sandro Morasca

**Abstract** Software systems continuously change for various reasons, such as adding new features, fixing bugs, or refactoring. Changes may either increase the source code complexity and disorganization, or help to reducing it. This paper empirically investigates the relationship of source code complexity and disorganization—measured using source code change entropy—with four factors, namely the presence of refactoring activities, the number of developers working on a source code file, the participation of classes in design patterns, and the different kinds of changes occurring on the system, classified in terms of their topics extracted from commit notes. We carried out an exploratory study on an interval of the life-time span of four open source systems, namely ArgoUML, Eclipse-JDT, Mozilla, and Samba, with the aim of analyzing the relationship between the source code change entropy and four factors: refactoring activities, number of contributors for a file, participation of classes in design patterns, and change topics. The study shows that (i) the change entropy decreases after refactoring, (ii) files changed by a higher number of

This paper is an extension of the paper "An Exploratory Study of Factors Influencing Change Entropy" (Canfora et al. 2010).

G. Canfora · M. Di Penta (✉)
Department of Engineering-RCOST, University of Sannio, Benevento, Italy
e-mail: dipenta@unisannio.it
URL: www.rcost.unisannio.it/mdipenta

G. Canfora
e-mail: canfora@unisannio.it
URL: www.gerardocanfora.net/

L. Cerulo
Department of Biological and Environmental Studies, University of Sannio, Benevento, Italy
e-mail: lcerulo@unisannio.it
URL: http://rcost.unisannio.it/cerulo

M. Cimitile
Department of Jurisprudence, Unitelma Sapienza, Napoli, Italy
e-mail: marta.cimitile@unitelma.it



[Canfora et al. EMSE 2014]

**Change entropy is higher when a change is scattered across many source files**

**The definition is directly related to the intuition that developers will have a harder work keeping track of changes that are performed across many source files**

Empir Software Eng (2014) 19:1–38
DOI 10.1007/s10664-012-9214-z

# How changes affect software entropy: an empirical study

**Gerardo Canfora · Luigi Cerulo · Marta Cimitile · Massimiliano Di Penta**

Published online: 14 July 2012

**Editor:** Sandro Morasca

**Abstract** Software systems continuously change for various reasons, such as adding new features, fixing bugs, or refactoring. Changes may either increase the source code complexity and disorganization, or help to reducing it. This paper empirically investigates the relationship of source code complexity and disorganization—measured using source code change entropy—with four factors, namely the presence of refactoring activities, the number of developers working on a source code file, the participation of classes in design patterns, and the different kinds of changes occurring on the system, classified in terms of their topics extracted from commit notes. We carried out an exploratory study on an interval of the life-time span of four open source systems, namely ArgoUML, Eclipse-JDT, Mozilla, and Samba, with the aim of analyzing the relationship between the source code change entropy and four factors: refactoring activities, number of contributors for a file, participation of classes in design patterns, and change topics. The study shows that (i) the change entropy decreases after refactoring, (ii) files changed by a higher number of

This paper is an extension of the paper "An Exploratory Study of Factors Influencing Change Entropy" (Canfora et al. 2010).

G. Canfora · M. Di Penta (✉)
Department of Engineering-RCOST, University of Sannio, Benevento, Italy
e-mail: dipenta@unisannio.it
URL: www.rcost.unisannio.it/mdipenta

G. Canfora
e-mail: canfora@unisannio.it
URL: www.gerardocanfora.net/

L. Cerulo
Department of Biological and Environmental Studies, University of Sannio, Benevento, Italy
e-mail: lcerulo@unisannio.it
URL: http://rcost.unisannio.it/cerulo

M. Cimitile
Department of Jurisprudence, Unitelma Sapienza, Napoli, Italy
e-mail: marta.cimitile@unitelma.it



[Canfora et al. EMSE 2012]

## Empirical study

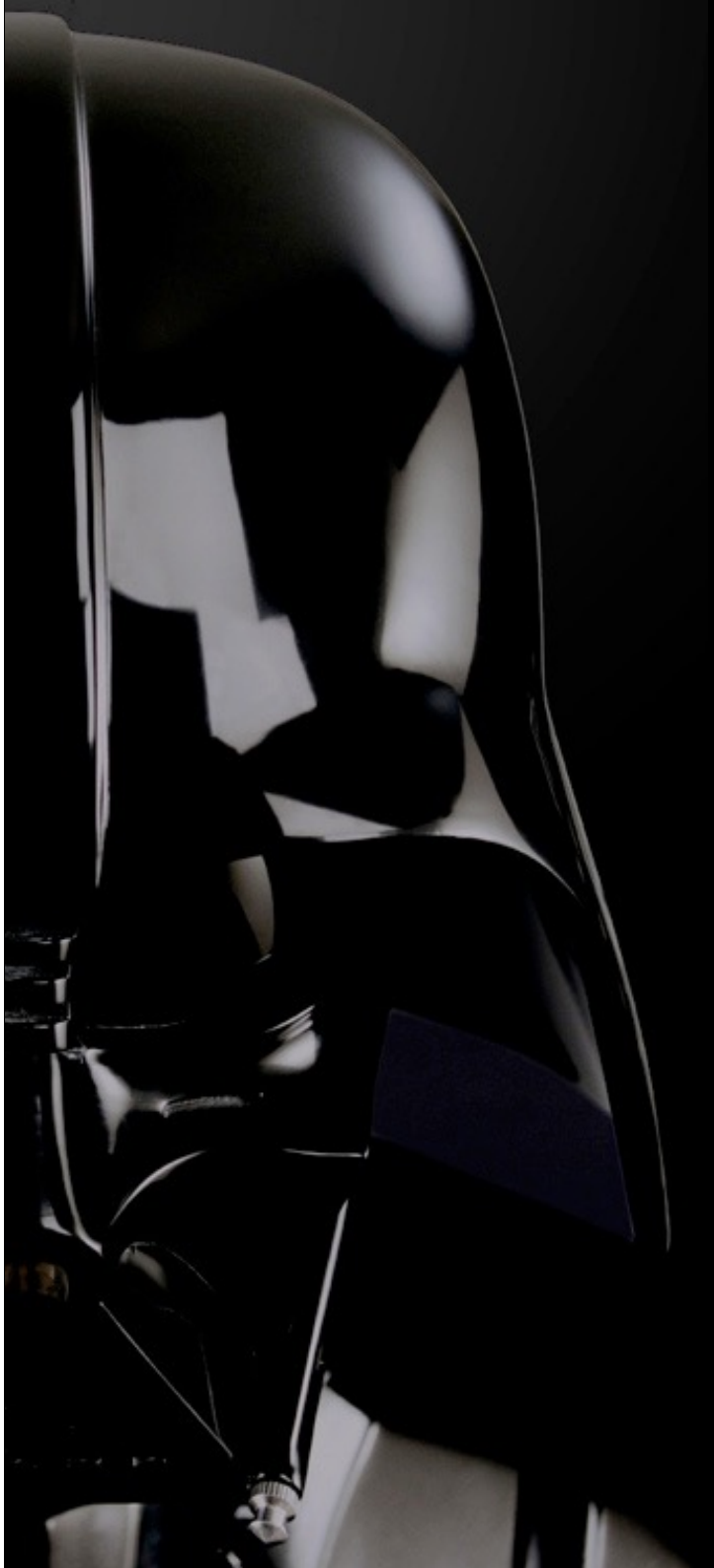
ArgoUML [11 years]
Eclipse-JDT [10 years]
Mozilla [13 years]
Samba [8 years]

## Identifying refactoring operations

by inspecting CVS/SVN commit notes and mining keywords likely describing refactoring activities [Ratzinger et al., MSR 2008]

## Results

The mean change entropy after refactoring always decreases



# Refactoring

Is there a dark side?

# When does a Refactoring Induce Bugs?

2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation

When does a Refactoring Induce Bugs?
An Empirical Study

Gabriele Bavota[1], Bernardino De Carluccio[1], Andrea De Lucia[1]
Massimiliano Di Penta[2], Rocco Oliveto[3], Orazio Strollo[1]
[1]University of Salerno, Fisciano (SA), Italy
[2]University of Sannio, Benevento, Italy
[3]University of Molise, Pesche (IS), Italy
gbavota@unisa.it, bernardino.decarluccio@gmail.com, adelucia@unisa.it
dipenta@unisannio.it, rocco.oliveto@unimol.it, orazio.strollo@hotmail.com

**Abstract**—Refactorings are—as defined by Fowler—behavior preserving source code transformations. Their main purpose is to improve maintainability or comprehensibility, or also reduce the code footprint if needed. In principle, refactorings are defined as simple operations so that are "unlikely to go wrong" and introduce faults. In practice, refactoring activities could have their risks, as other changes.

This paper reports an empirical study carried out on three Java software systems, namely Apache Ant, Xerces, and ArgoUML, aimed at investigating to what extent refactoring activities induce faults. Specifically, we automatically detect (and then manually validate) 15,008 refactoring operations (of 52 different kinds) using an existing tool (Ref-Finder). Then, we use the SZZ algorithm to determine whether it is likely that refactorings induced a fault.

Results indicate that, while some kinds of refactorings are unlikely to be harmful, others, such as refactorings involving hierarchies (e.g., pull up method), tend to induce faults very frequently. This suggests more accurate code inspection or testing activities when such specific refactorings are performed.

**Index Terms**—Refactoring, Fault-inducing changes, Mining software repositories, Empirical Studies.

## I. Introduction

Software systems are continuously subject to maintenance tasks to introduce new features or fix bugs [1]. Very often such activities are performed in an undisciplined manner due to strict time constraints, to lack of maintainability, or to the limited knowledge some developers have of the system design [2]. As a result, the code underlying structure, and therefore the related design, tend to deteriorate.

This phenomenon was defined as "software aging" by Parnas [3], and was also described in the law of increasing complexity by Lehman [1]. Some researchers measured the phenomenon in terms of change entropy [4], [5], while others defined "antipatterns", i.e., recurring cases of poor design choices occurring as a consequence of aging, or when the software is not properly designed from the beginning. Classes doing too much (*God classes* or *Blobs*), poorly structured code (*Spaghetti code*), or *Long Message Chains* used to develop a certain feature are only few examples of antipatterns that plague software systems [2].

In order to mitigate the above described issues, software systems are, time to time, subject to improvement activities, aimed at enhancing the code and design structure. Such activities are often referred to as *refactoring*. Refactoring is defined by Fowler [3] as "*a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior*". The aim of refactoring is to improve the structure of source code—and consequently of the system design—whenever its structure may possibly lead to maintainability or comprehensibility problems. Fowler's catalogue [6] comprises a set of 93 refactorings, aimed at dealing with different antipatterns in source code, such as, extracting a class from a Blob, pulling up a method from a subclass onto a superclass, or modifying the navigability of an association between two classes.

In theory, a refactoring should not change the behavior of a software system, but only help in improving some of its non-functional attributes. In practice, a refactoring might be risky as any other change occurring in a system, causing possible bug introductions. Indeed, a recent study [10] showed that even automated refactoring as performed by Integrated Development Environments could be fault-prone as well.

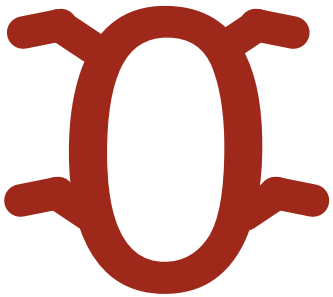
While there are attempts to investigate the relation between some refactorings and fault-proneness [8], [9] or change entropy [7], to the best of our knowledge there is no study aimed at thoroughly investigating whether a wide set of (even undocumented) refactorings occurred in a software system during its evolution induced bugs, and what kind of refactorings might induce more bugs than others.

In this paper we report an empirical study aimed at investigating to what extent refactoring induces bug fixes in software systems. We use an existing tool, namely Ref-Finder [11], to automatically detect refactoring operations of 52 different types on 63 releases of three Java software systems, Apache Ant[1], ArgoUML[2], and Xerces-J[3]. Of the 15,008 refactoring operations detected by the tool, 12,922 operations have been manually validated as actually refactorings. Then, we use the SZZ algorithm [12], [13] to determine whether the 12,922

[1]http://ant.apache.org/
[2]http://argouml.tigris.org/
[3]http://xerces.apache.org/xerces-j/

978-0-7695-4793-1/12 $26.00 © 2012 IEEE
DOI 10.1109/SCAM.2012.20

194



Bavota et al. [SCAM 2012]

# 63 releases of 3 systems

**Apache Ant  ArgoUML  Xerces**

# 15,008 refactorings of 52 different types


MANUALLY
VALIDATED
12,922
refactorings of 52
different types


what about
bugs?

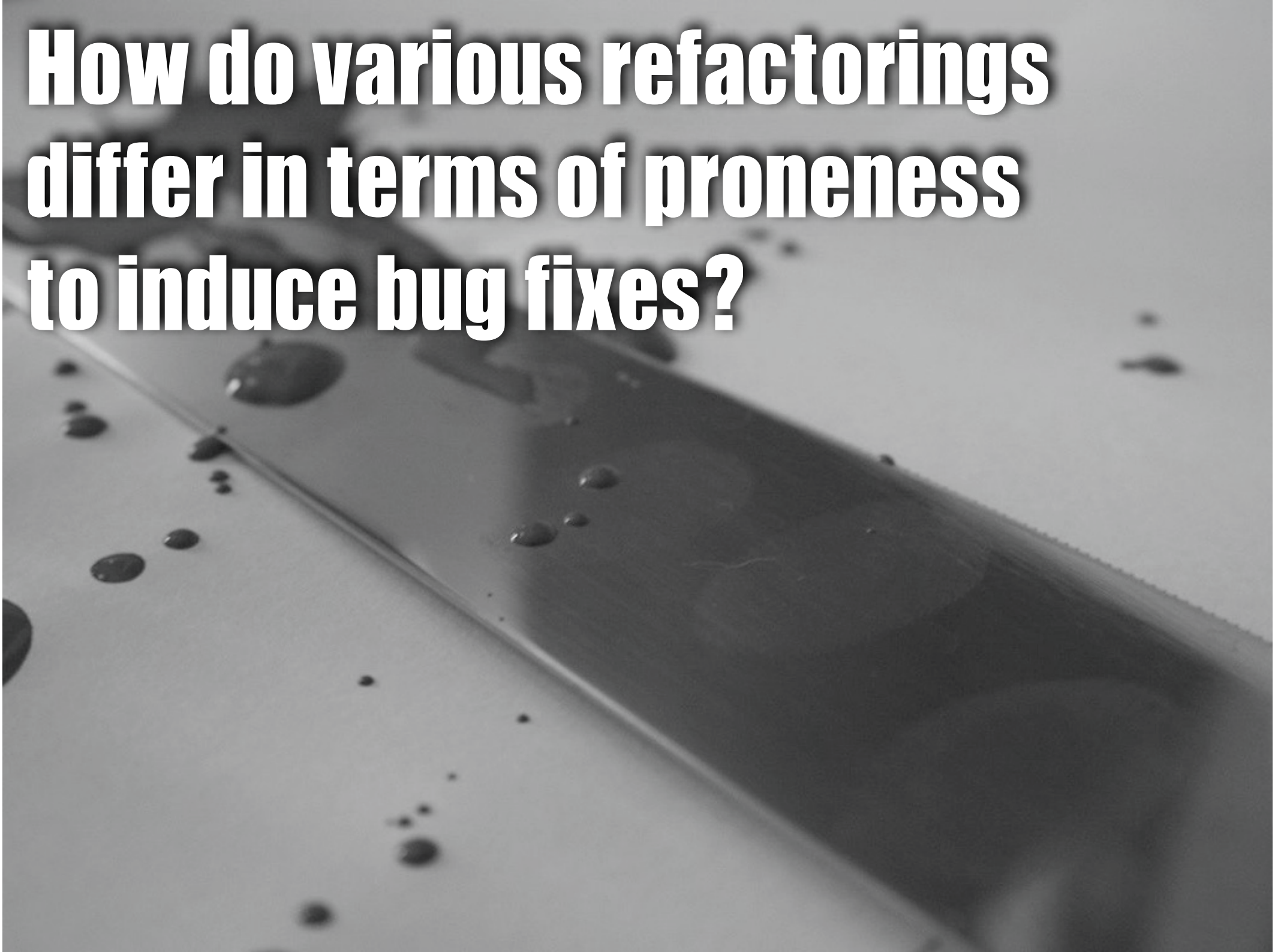

# we considered only fixed bugs

# To what extent do refactorings induce bug fixes?

**In all 11 releases classes involved in refactorings have a higher chance of being involved in bug-fixes**

**23** times higher on average

**158** times in the worst case


How do various refactorings
differ in terms of proneness
to induce bug fixes?

For 52 different kinds of refactorings, we compared the percentage of refactored classes for which refactorings induced a bug fix

13% median percentage of fault-prone refactored classes

we identified very dangerous refactorings

# ...some numbers...

**40%**

The percentage of classes refactored through **pull up method** or **extract subclass** subject to bug-fixing

**20%**

**Inline Temp**
**Replace Method With Method Object**
**Extract Method**

# When is Refactoring Performed?

The Journal of Systems and Software 107 (2015) 1–14

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

# An experimental investigation on the innate relationship between quality and refactoring

Gabriele Bavota [a,*], Andrea De Lucia [b], Massimiliano Di Penta [c], Rocco Oliveto [d], Fabio Palomba [b]

[a] Free University of Bozen-Bolzano, Bolzano, Italy
[b] University of Salerno, Fisciano (SA), Italy
[c] University of Sannio, Benevento, Italy
[d] University of Molise, Pesche (IS), Italy

ARTICLE INFO

*Article history:*
Received 8 April 2015
Revised 8 May 2015
Accepted 12 May 2015
Available online 21 May 2015

*Keywords:*
Refactoring
Code smells
Empirical study

ABSTRACT

Previous studies have investigated the reasons behind refactoring operations performed by developers, and proposed methods and tools to recommend refactorings based on quality metric profiles, or on the presence of poor design and implementation choices, i.e., code smells. Nevertheless, the existing literature lacks observations about the relations between metrics/code smells and refactoring activities performed by developers. In other words, the characteristics of code components increasing/decreasing their chances of being object of refactoring operations are still unknown. This paper aims at bridging this gap. Specifically, we mined the evolution history of three Java open source projects to investigate whether refactoring activities occur on code components for which certain indicators—such as quality metrics or the presence of smells as detected by tools—suggest there might be need for refactoring operations. Results indicate that, more often than not, quality metrics do not show a clear relationship with refactoring. In other words, refactoring operations are generally focused on code components for which quality metrics do not suggest there might be need for refactoring operations. Finally, 42% of refactoring operations are performed on code entities affected by code smells. However, only 7% of the performed operations actually remove the code smells from the affected class.



## 1. Introduction

Refactoring has been defined by Fowler as "*the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure*" (Fowler et al., 1999). This definition entails a strong relationship between refactoring and internal software quality, i.e., refactoring improves software quality (*improves the software internal structure*). This has motivated research on bad smell and antipattern detection and on the identification of refactoring oppotunities (Bavota et al., 2013a; Boussaa et al., 2013; Fokaefs et al., 2011; Kessentini et al., 2010; Moha et al., 2010; Palomba et al., 2015; Tsantalis and Chatzigeorgiou, 2009).

However, whether refactoring is actually guided by poor design has not been empirically evaluated enough. Thus, this assumption still remains—for some aspects—a common wisdom that has generated controversial positions (Kim et al., 2012). Specifically, there are no studies that **quantitatively** analyze which are the quality characteristics of the source code increasing their likelihood of being subject of refactoring operations. To the best of our knowledge, the available empirical evidence is based on two surveys performed with developers trying to understand the reasons why developers perform refactoring operations (Kim et al., 2012; Wang, 2009).

In addition, concerning the improvement of the internal quality of software, empirical studies have only shown that generally refactoring operations improve the values of quality metrics (Kataoka et al., 2002; Leitch and Stroulia, 2003; Moser et al., 2006; Ratzinger et al., 2005; Shatnawi and Li, 2011), while the effectiveness of refactoring in removing design flaws (such as code smells) is still unknown.

In order to fill this gap, we use an existing tool, namely Ref-Finder (Prete et al., 2010), to automatically detect refactoring operations of 52 different types on 63 releases of three Java software systems, namely Apache Ant,[1] ArgoUML,[2] and Xerces-J.[3] Since Ref-Finder can identify some false positives, we manually analyzed the 15,008 refactoring operations detected by the tool. Among them, 2086 were

* Corresponding author. Tel.: +39 333 594 4151.
*E-mail address:* gabriele.bavota@unibz.it (G. Bavota).

http://dx.doi.org/10.1016/j.jss.2015.05.024
0164-1212/

[1] http://ant.apache.org/.
[2] http://argouml.tigris.org/.
[3] http://xerces.apache.org/xerces-j/.

Bavota et al. [JSS 2015]

3 open source systems

11 quality metrics

11 bad smells

12,922 refactoring operations manually validated

# Studying if quality metrics / bad smells "induce" refactoring operations

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%

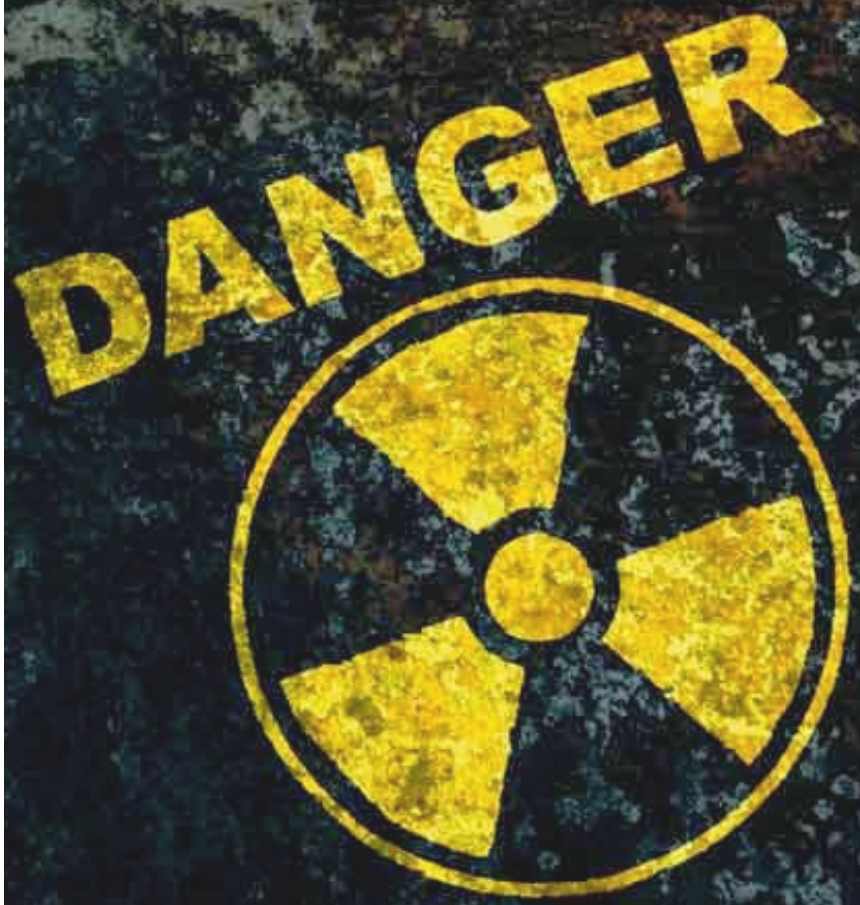
of refactoring operations are performed on code entities affected by code smells.

However, often refactoring fails in removing code smells!

Only 7%

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

# Some refactoring operations even introduce code smells!




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

## When and Why Your Code Starts to Smell Bad

Michele Tufano*, Fabio Palomba†, Gabriele Bavota‡, Rocco Oliveto§,
Massimiliano Di Penta¶, Andrea De Lucia†, Denys Poshyvanyk*
*The College of William and Mary, Williamsburg, VA, USA - †University of Salerno, Fisciano (SA), Italy
‡Free University of Bozen-Bolzano, Italy - §University of Molise, Pesche (IS), Italy
¶University of Sannio, Benevento, Italy

***Abstract*—In past and recent years, the issues related to managing technical debt received significant attention by researchers from both industry and academia. There are several factors that contribute to technical debt. One of these is represented by code bad smells, *i.e.*, symptoms of poor design and implementation choices. While the repercussions of smells on code quality have been empirically assessed, there is still only anecdotal evidence on *when* and *why* bad smells are introduced. To fill this gap, we conducted a large empirical study over the change history of 200 open source projects from different software ecosystems and investigated when bad smells are introduced by developers, and the circumstances and reasons behind their introduction. Our study required the development of a strategy to identify smell-introducing commits, the mining of over 0.5M commits, and the manual analysis of 9,164 of them (*i.e.*, those identified as *smell-introducing*). Our findings mostly contradict common wisdom stating that smells are being introduced during evolutionary tasks. In the light of our results, we also call for the need to develop a new generation of recommendation systems aimed at properly planning smell refactoring activities.**

### I. Introduction

Technical debt is a metaphor introduced by Cunningham to indicate *"not quite right code which we postpone making it right"* [18]. The metaphor explains well the trade-offs between delivering the most appropriate but still immature product, in the shortest time possible [12], [18], [27], [31], [42]. While the repercussions of "technical debt" on software quality have been empirically proven, there is still noticeable lack of empirical evidence related to how, when, and why various forms of technical debt occur in software projects [12]. This represents an obstacle for an effective and efficient management of technical debt.

Bad code smells (shortly "code smells" or "smells"), *i.e.*, symptoms of poor design and implementation choices [20], represent one important factor contributing to technical debt, and possibly affecting the maintainability of a software system [27]. In the past and, most notably, in recent years, several studies investigated the relevance that code smells have for developers [37], [50], the extent to which code smells tend to remain in a software system for long periods of time [3], [15], [32], [40], as well as the side effects of code smells, such as increase in change- and fault-proneness [25], [26] or decrease of software understandability [1] and maintainability [43], [49], [48]. The research community has been also actively developing approaches and tools for detecting smells [11], [34], [36], [44], [33], and, whenever possible, triggering refactoring operations. Such tools rely on different types of analysis techniques, such as constraint-based reasoning over metric values [33], [34], static code analysis [44], or analysis of software changes [36]. While these tools provide relatively accurate and complete identification of a wide variety of smells, most of them work by "taking a snapshot" of the system or by looking at recent changes, hence providing a snapshot-based recommendation to the developer. Hence, they do not consider the circumstances that could have caused the smell introduction. In order to better support developers in planning actions to improve design and source code quality, it is imperative to have a contextualized understanding of the circumstances under which particular smells occur. However, to the best of our knowledge, there is no comprehensive empirical investigation into *when* and *why* code smells are introduced in software projects. Common wisdom suggests that urgent maintenance activities and pressure to deliver features while prioritizing time-to-market over code quality are often the causes of such smells. Generally speaking, software evolution has always been considered as one of the reasons behind "software aging" [38] or "increasing complexity" [28][35][47]. Broadly speaking, smells can also manifest themselves not only in the source code but also in software lexicons [29], [4], and can even affect other types of artifacts, such as spreadsheets [22], [23] or test cases [9].

In this paper we fill the void in terms of our understanding of code smells, reporting the results of a large-scale empirical study conducted on the evolution history of 200 open source projects belonging to three software ecosystems, namely Android, Apache and Eclipse. The study aimed at investigating (i) *when* smells are introduced in software projects, and (ii) *why* they are introduced, *i.e.*, under what circumstances smell introductions occur and who are the developers responsible for introducing smells. To address these research questions, we developed a metric-based methodology for analyzing the evolution of code entities in change histories of software projects to determine when code smells start manifesting themselves and whether this happens suddenly (*i.e.*, because of a pressure to quickly introduce a change), or gradually (*i.e.*, because of medium-to-long range design decisions). We mined over 0.5M commits and we manually analyzed 9,164 of those that were classified as *smell-introducing*. We are unaware of any published technical debt, in general, and code smell study, in particular, of comparable size. The results achieved allowed us to report quantitative and qualitative evidence on when and

* Michele Tufano and Denys Poshyvanyk from W&M were partially supported via NSF CCF-1253837 and CCF-1218129 grants.
† Fabio Palomba is partially funded by the University of Molise.

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.59

# Open Issues

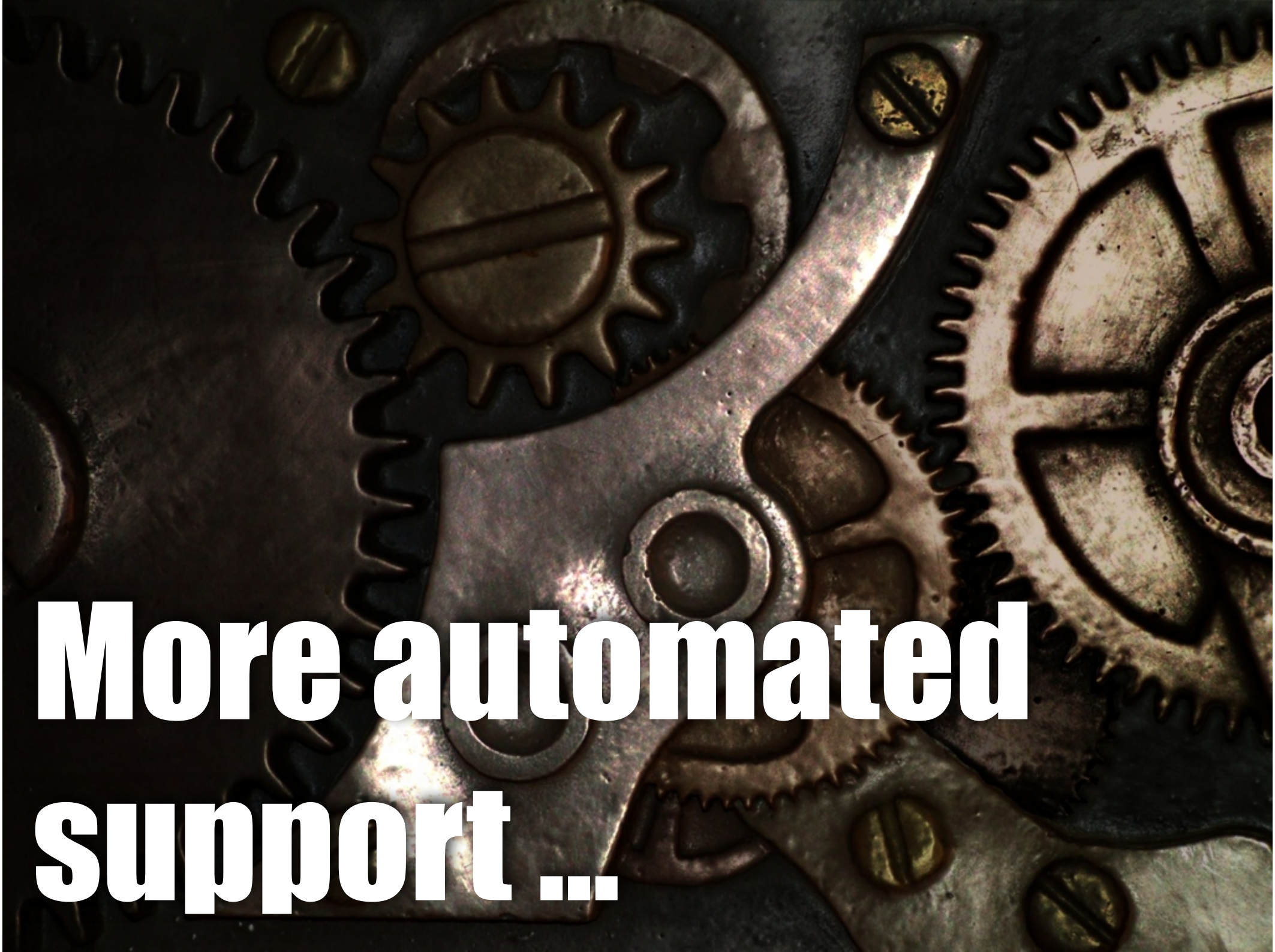
# Convincing Managers

# Let's talk about
# Technical Debts

*"Not quite right code which we postpone making it right"*

W. Cunningham
"The WyCash portfolio management system,"
OOPS 1993.

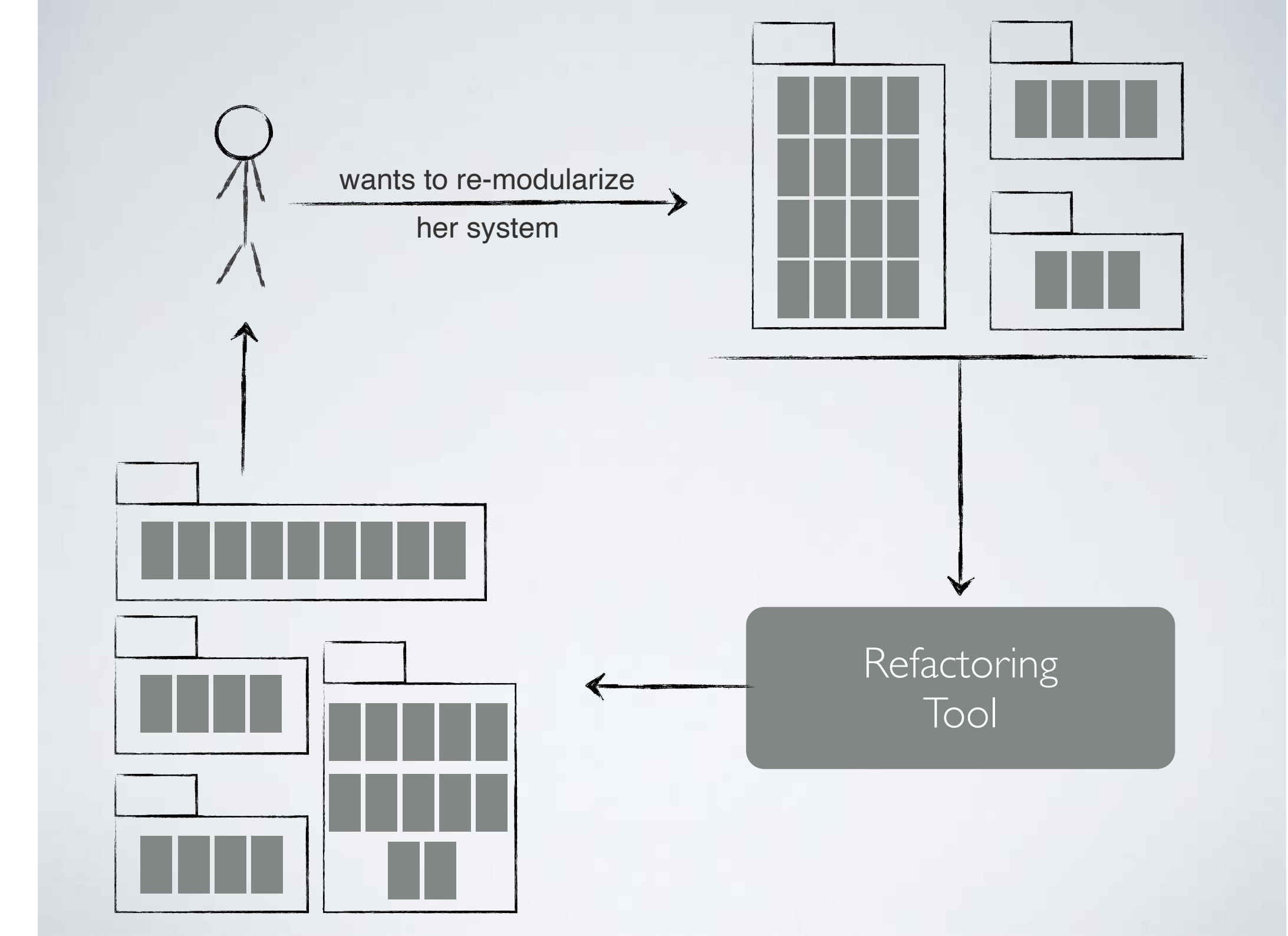
# Avoiding Risks

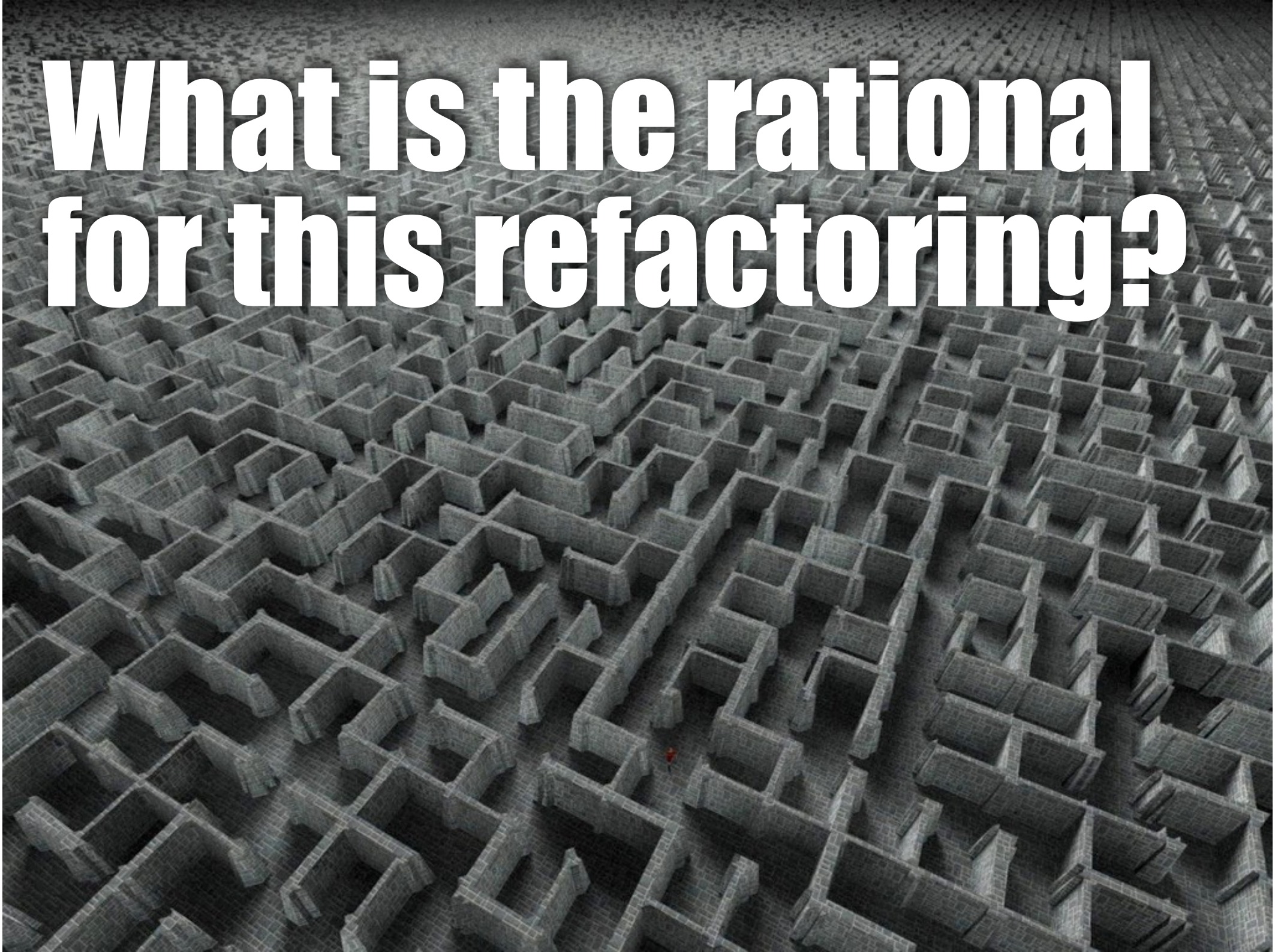

More automated
support ...


... preserving
system behavior

# Convincing Developers

# Usability of refactoring tools


wants to re-modularize
her system
Refactoring
Tool


What is the rational
for this refactoring?

A

**Improving Software Modularization via Automated Analysis of Latent Topics and Dependencies**

GABRIELE BAVOTA, University of Salerno
MALCOM GETHERS, University of Maryland, Baltimore County
ROCCO OLIVETO, University of Molise
DENYS POSHYVANYK, The College of William and Mary
ANDREA DE LUCIA, University of Salerno

Oftentimes, during software maintenance the original program modularization decays, thus reducing its quality. One of the main reasons for such architectural erosion is suboptimal placement of source code classes in software packages. To alleviate this issue, we propose an automated approach to help developers improve the quality of software modularization. Our approach analyzes underlying latent topics in source code as well as structural dependencies to recommend (and explain) refactoring operations aiming at moving a class to a more suitable package. The topics are acquired via Relational Topic Models (RTM), a probabilistic topic modeling technique. The resulting tool, coined as *R3* (Rational Refactoring via RTM), has been evaluated in two empirical studies. The results of the first study conducted on nine software systems indicate that *R3* provides a coupling reduction from 10% to 30% among the software modules. The second study with 62 developers confirms that *R3* is able to provide meaningful recommendations (and explanations) for move class refactoring. Specifically, more than 70% of the recommendations where considered meaningful from a functional point of view.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms: Documentation, Management

Additional Key Words and Phrases: Software Modularization, Refactoring, Relational Topic Modeling, Empirical Studies, Recommendation System

**1. INTRODUCTION**

In the software life-cycle the change is the rule and not the exception [Lehman 1980]. A key point for sustainable program evolution is to tackle software complexity. In Object-Oriented (OO) systems, classes are the primary decomposition mechanism, which group together data and operations to reduce complexity. Higher level programming constructs, such as packages, group semantically and structurally related classes aiming at supporting the replacement of specific parts of a system without impacting the complete system. A well modularized system eases the understanding, maintenance, test, and evolution of software systems [DeRemer and Kron 1976].

Author's addresses: Gabriele Bavota and Andrea De Lucia, University of Salerno, Fisciano (SA), Italy; Rocco Oliveto, University of Molise, Pesche (IS), Italy; Malcom Gethers, University of Maryland, Baltimore County, Baltimore, MD 21250, USA; Denys Poshyvanyk, The College of William and Mary, Williamsburg, VA 23185, USA.


DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000



[Bavota et al. TOSEM 2014]

**A first attempt to automatically generate refactoring explanations**

**What about using Natural Language techniques?**

**Final goal**

**build a plug-in for refactoring tools that, analyzing code before and after refactoring explain the rational**

# Interactive Tools?

## Putting the Developer in-the-Loop: An Interactive GA for Software Re-modularization

Gabriele Bavota[1], Filomena Carnevale[1], Andrea De Lucia[1],
Massimiliano Di Penta[2], and Rocco Oliveto[3]

[1] University of Salerno, Via Ponte don Melillo, 84084 Fisciano (SA), Italy
[2] University of Sannio, Palazzo ex Poste, Via Traiano, 82100 Benevento, Italy
[3] University of Molise, Contrada Fonte Lappone, 86090 Pesche (IS), Italy
{gbavota,adelucia}@unisa.it, flmn.carnevale@gmail.com,
dipenta@unisannio.it, rocco.oliveto@unimol.it

**Abstract.** This paper proposes the use of Interactive Genetic Algorithms (IGAs) to integrate developer's knowledge in a re-modularization task. Specifically, the proposed algorithm uses a fitness composed of automatically-evaluated factors—accounting for the modularization quality achieved by the solution—and a human-evaluated factor, penalizing cases where the way re-modularization places components into modules is considered meaningless by the developer.

The proposed approach has been evaluated to re-modularize two software systems, SMOS and GESA. The obtained results indicate that IGA is able to produce solutions that, from a developer's perspective, are more meaningful than those generated using the full-automated GA. While keeping feedback into account, the approach does not sacrifice the modularization quality, and may work requiring a very limited set of feedback only, thus allowing its application also for large systems without requiring a substantial human effort.

### 1 Introduction

Software is naturally subject to change activities aiming at fixing bugs or introducing new features. Very often, such activities are conducted within a very limited time frame, and with a limited availability of software design documentation. Change activities tend to "erode" the original design of the system. Such a design erosion mirrors a reduction of the cohesiveness of a module, the increment of the coupling between various modules and, therefore, makes the system harder to be maintained or, possibly, more fault-prone [8]. For this reason, various automatic approaches, aimed at supporting source code re-modularization, have been proposed in literature (see e.g., [11,14,20]. The underlying idea of such approaches is to (i) group together in a module highly cohesive source code components, where the cohesiveness is measured in terms of intra-module links; and (ii) reduce the coupling between modules, where the coupling is measured in terms of inter-module dependencies. Such approaches use various techniques,

G. Fraser (Ed.): SSBSE 2012, LNCS 7515, pp. 75–89, 2012.


Bavota et al. [SSBSE 2012]

# Using smell intensity to improve Bug Prediction Models


# 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 bugginess 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.

# Test Smells and Refactoring of Test Code

# How about smells related to other non functional requirements ?

# Particularly relevant for mobile applications


Any Questions ?


**Andrea De Lucia**
**Full Professor**
**University of Salerno**
**adelucia@unisa.it**