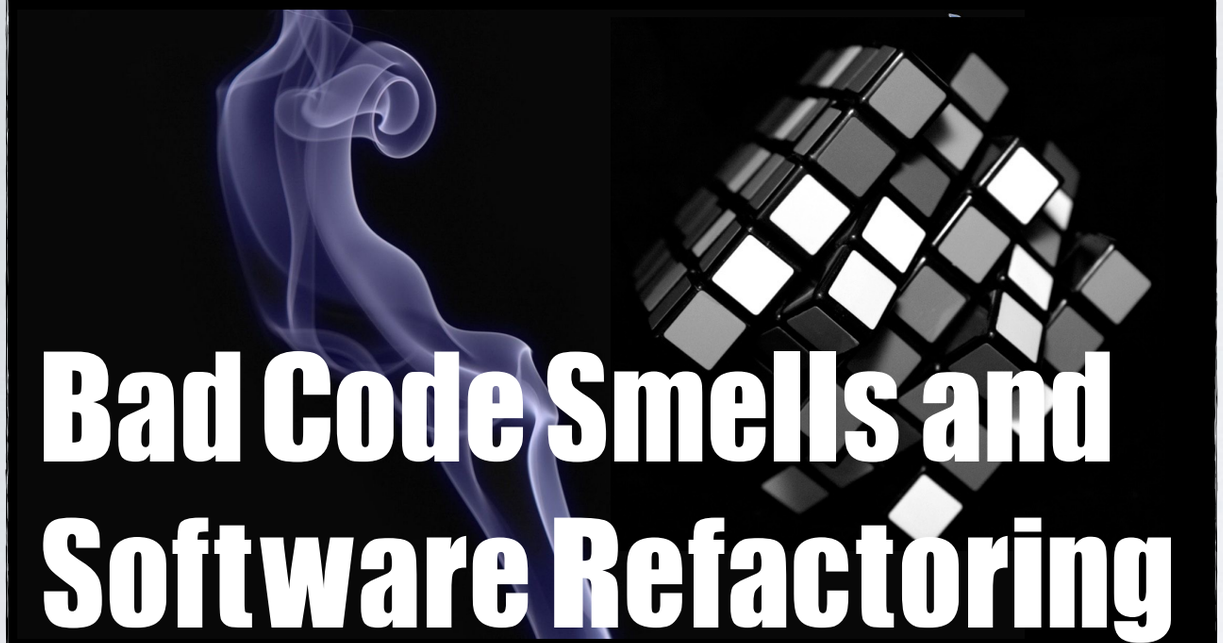
Code Smell Identification and Refactoring Automation: Challenges, Solutions, and Open Issues

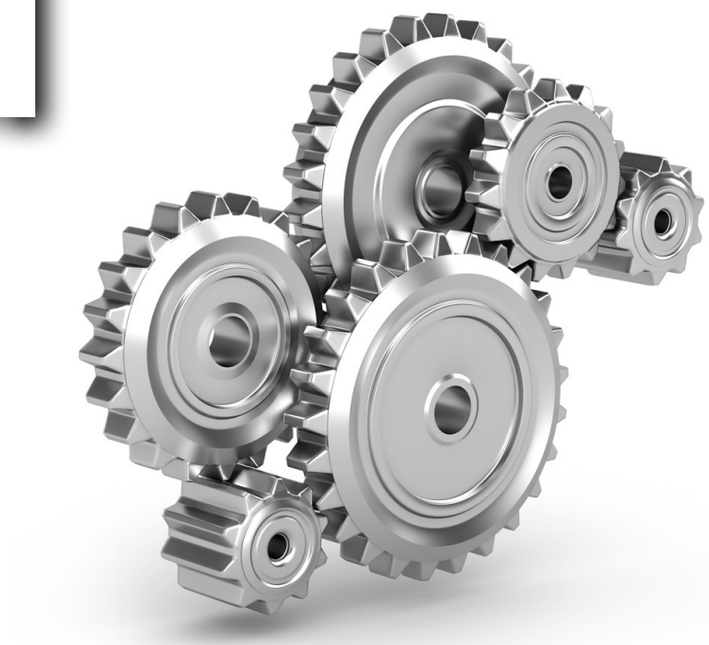
Fabio Palomba
University of Zurich
palomba@ifi.uzh.ch

Part I



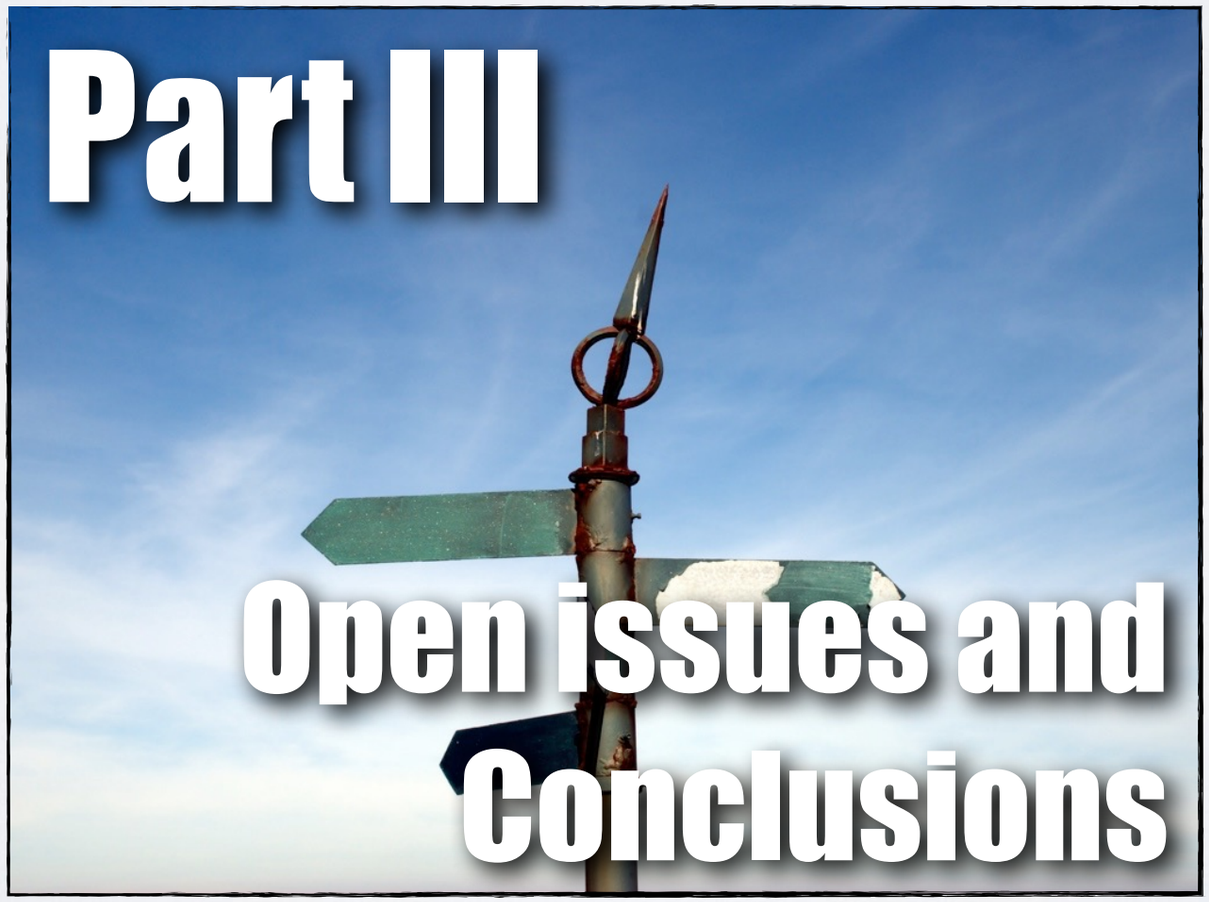
Bad Code Smells and Software Refactoring

Part II



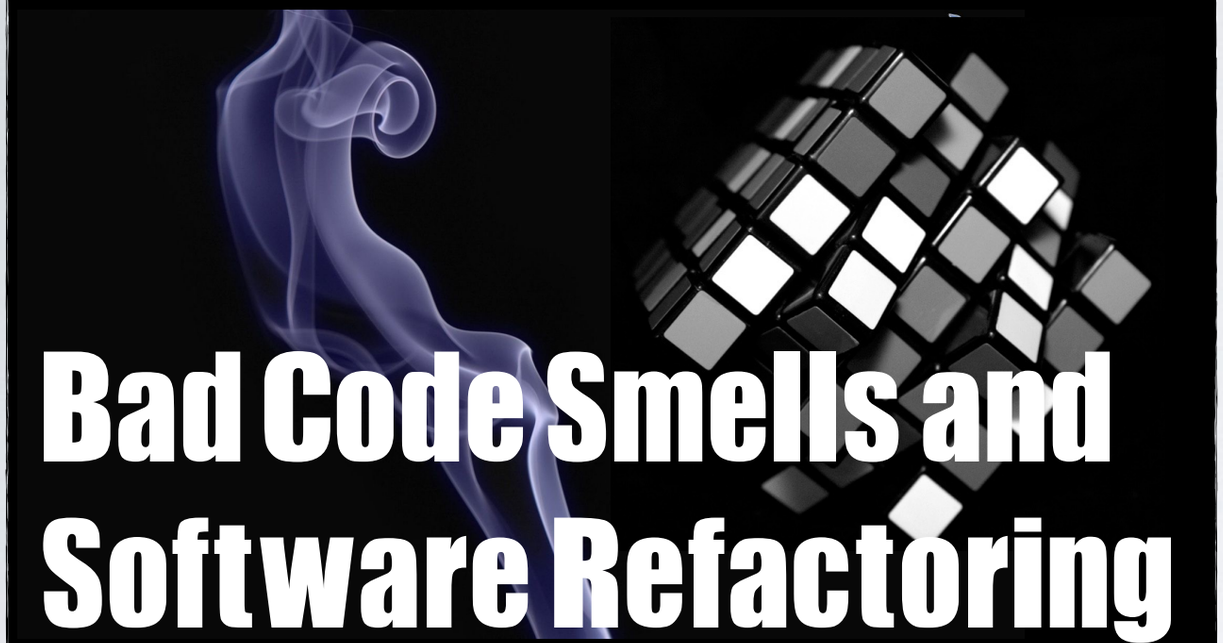
The refactoring process

Part III



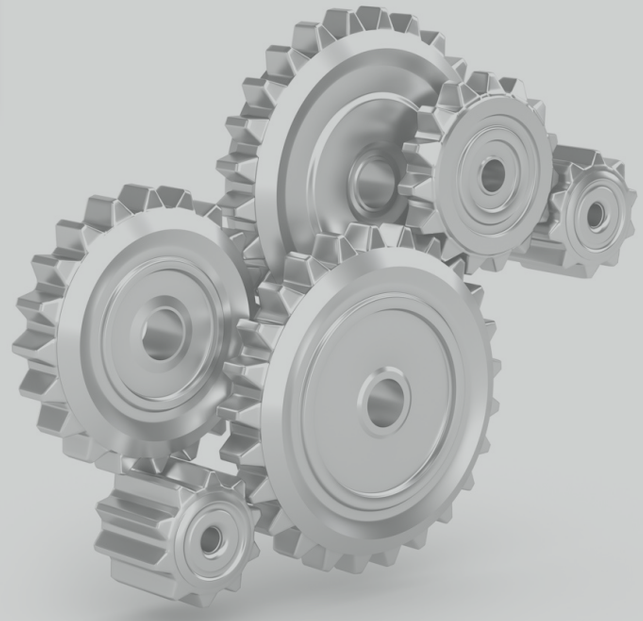
Open issues and Conclusions

Part I



Bad Code Smells and Software Refactoring

Part II

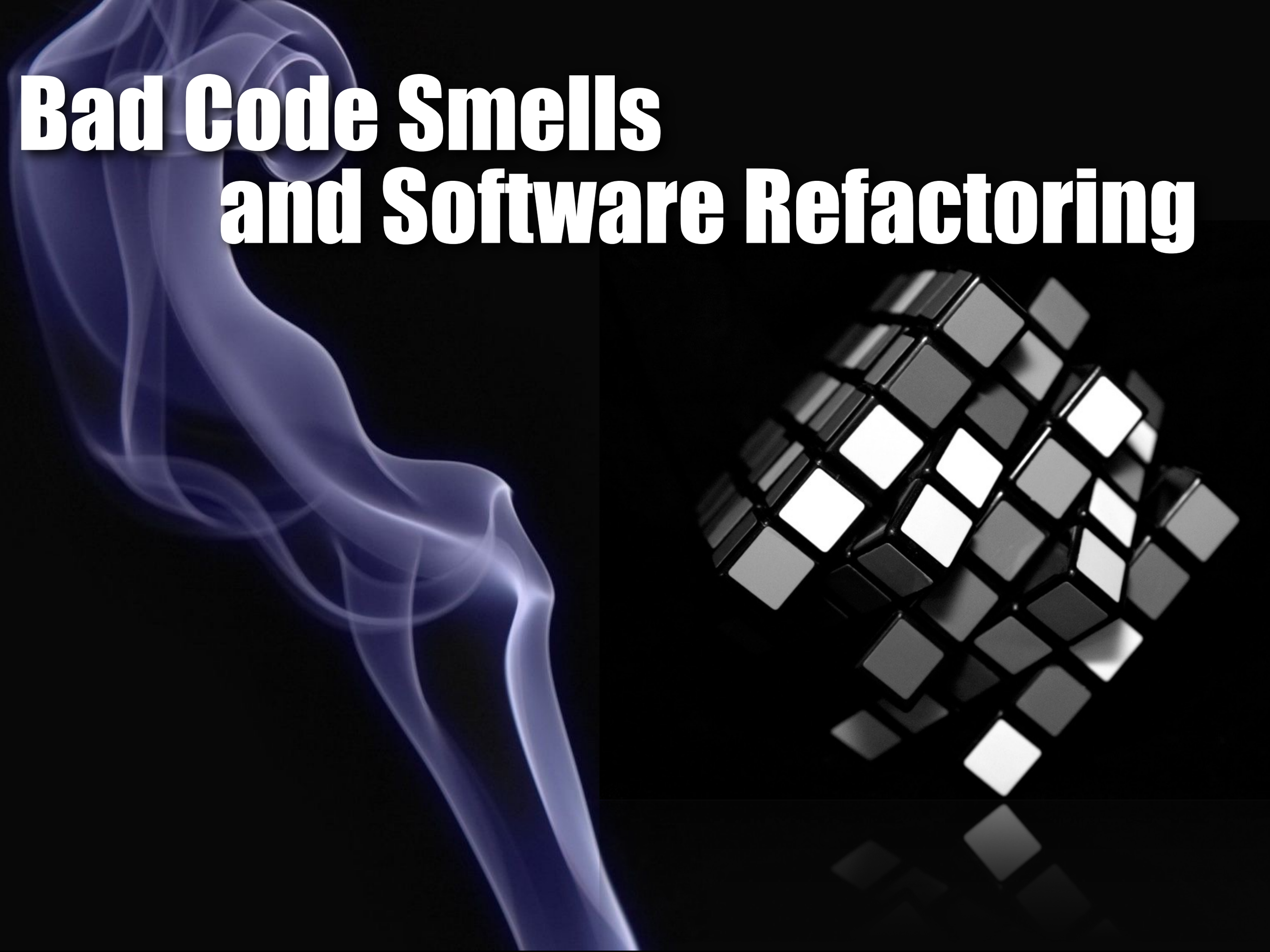


The refactoring process

Part III

Open issues and Conclusions

Bad Code Smells and Software Refactoring



Bad Code Smells

"Symptoms of poor design or implementation choices"

[Martin Fowler, 1999]

Software evolution

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

Low design quality ...

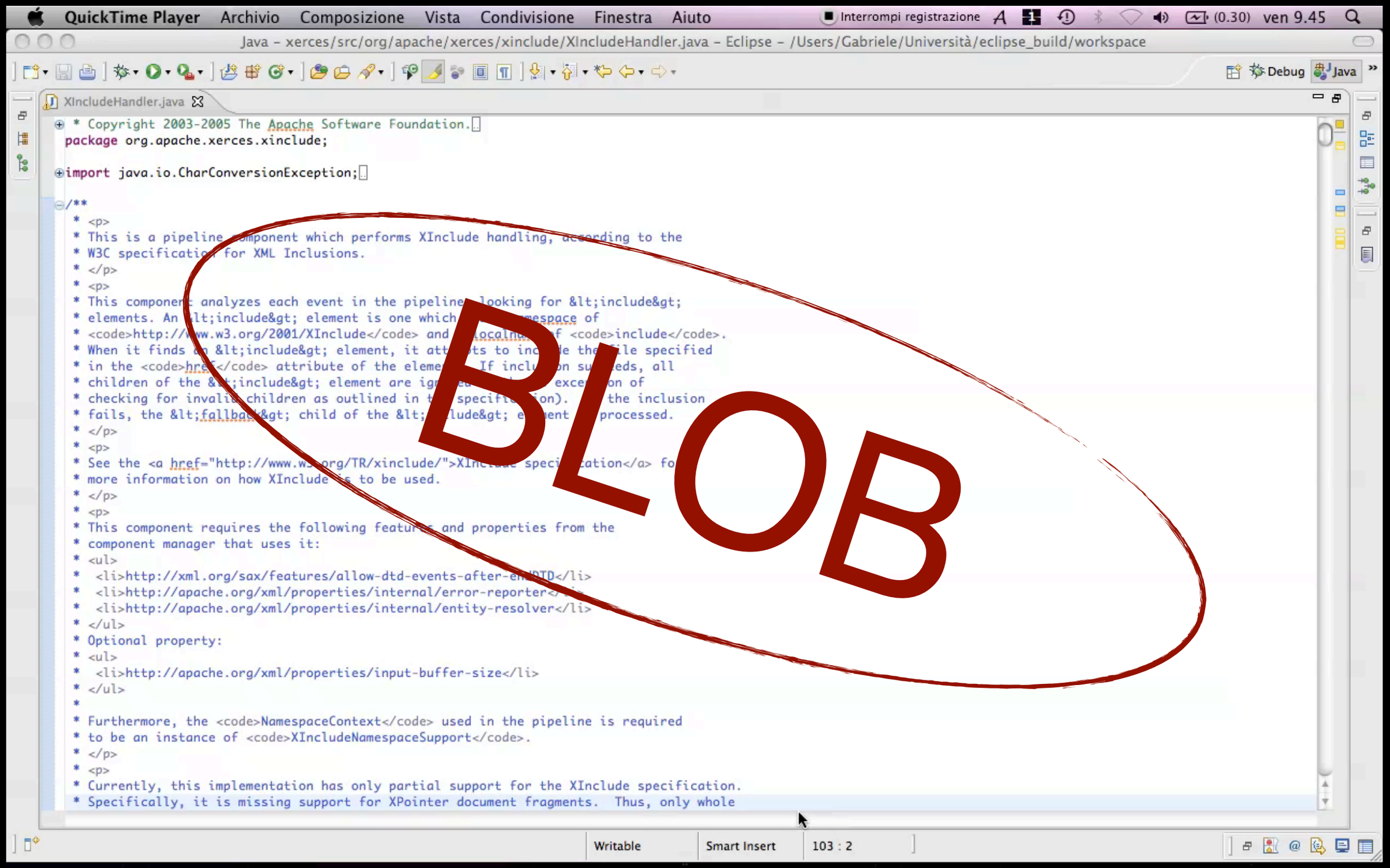
... has been associated with lower productivity, greater rework, and more significant efforts for developers

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
Interrompi registrazione
(0.30) ven 9.45
Java – xerces/src/org/apache/xerces/xinclude/XIncludeHandler.java – Eclipse – /Users/Gabriele/Università/eclipse_build/workspace
Debug
Java
XIncludeHandler.java
* Copyright 2003-2005 The Apache Software Foundation.
package org.apache.xerces.xinclude;
import java.io.CharConversionException;
/**
*
* This is a pipeline component which performs XInclude handling, according to the
* W3C specification for XML Inclusions.
*
*
* This component analyzes each event in the pipeline, looking for <include>
* elements. An <include> element is one which
* <code>http://www.w3.org/2001/XInclude</code> and
* When it finds an <include> element, it attempts to include the file specified
* in the <code>href</code> attribute of the element. If inclusion succeeds, all
* children of the <include> element are ignored, with exception of
* checking for invalid children as outlined in the specification). If the inclusion
* fails, the <fallback> child of the <include> element is processed.
*
*
* See the XInclude specification for
* more information on how XInclude is to be used.
*
*
* This component requires the following features and properties from the
* component manager that uses it:
*
* http://xml.org/sax/features/allow-dtd-events-after-endDTD
* http://apache.org/xml/properties/internal/error-reporter
* http://apache.org/xml/properties/internal/entity-resolver
*
* Optional property:
*
* http://apache.org/xml/properties/input-buffer-size
*
*
* Furthermore, the <code>NamespaceContext</code> used in the pipeline is required
* to be an instance of <code>XIncludeNamespaceSupport</code>.
*
*
* Currently, this implementation has only partial support for the XInclude specification.
* Specifically, it is missing support for XPointer document fragments. Thus, only whole
Writable
Smart Insert
103 : 2
BLOB

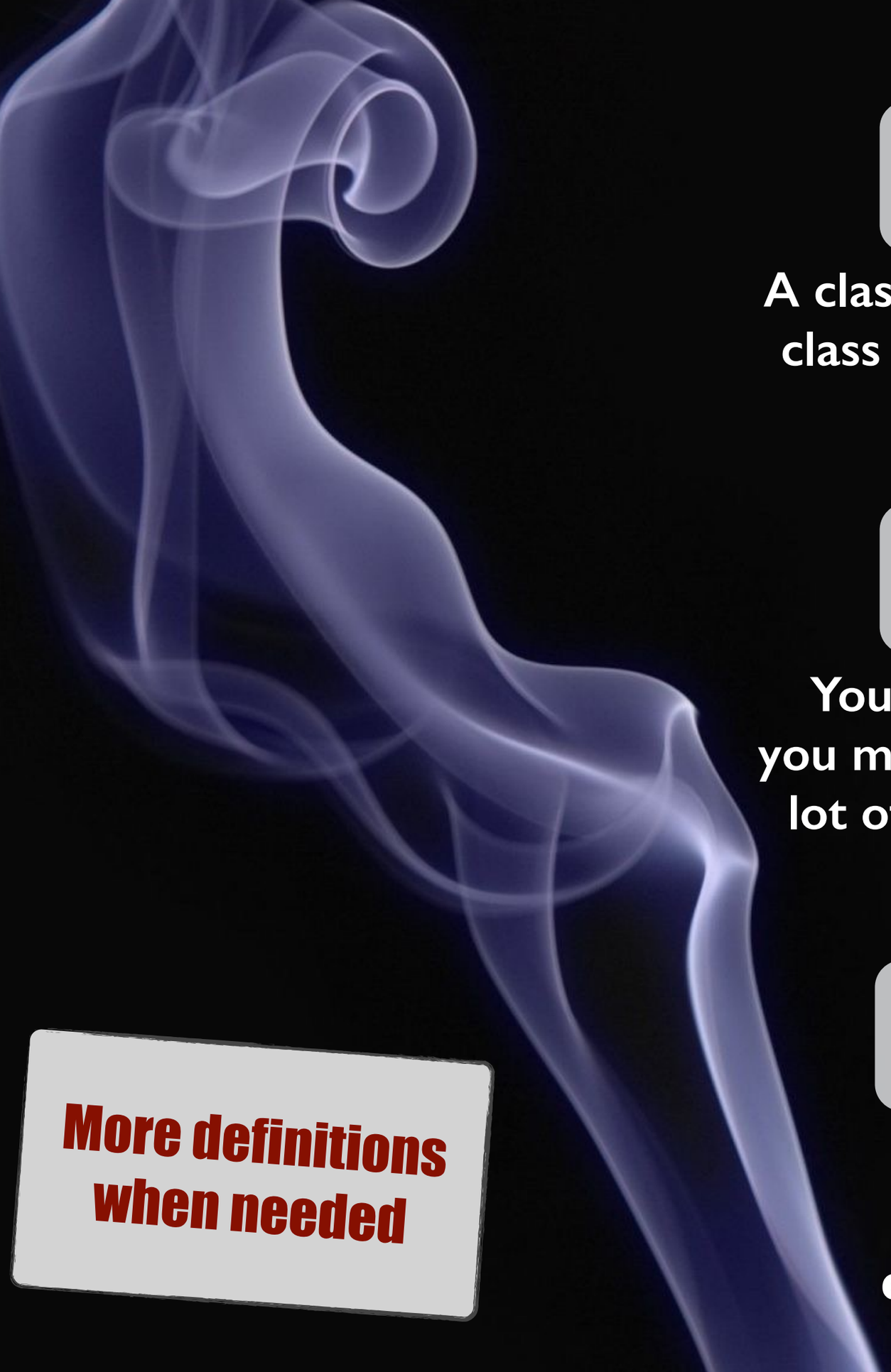
Blob (or God Class)

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

[Martin Fowler]

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

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

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

I. Introduction

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

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

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

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

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

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

Bad Smells hinder code comprehensibility

[Abbes et al. CSMR 2011]

Negative Impact of Bad Smells

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
© Springer Science+Business Media, LLC 2011
Editor: Jim Whitehead

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

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

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

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

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

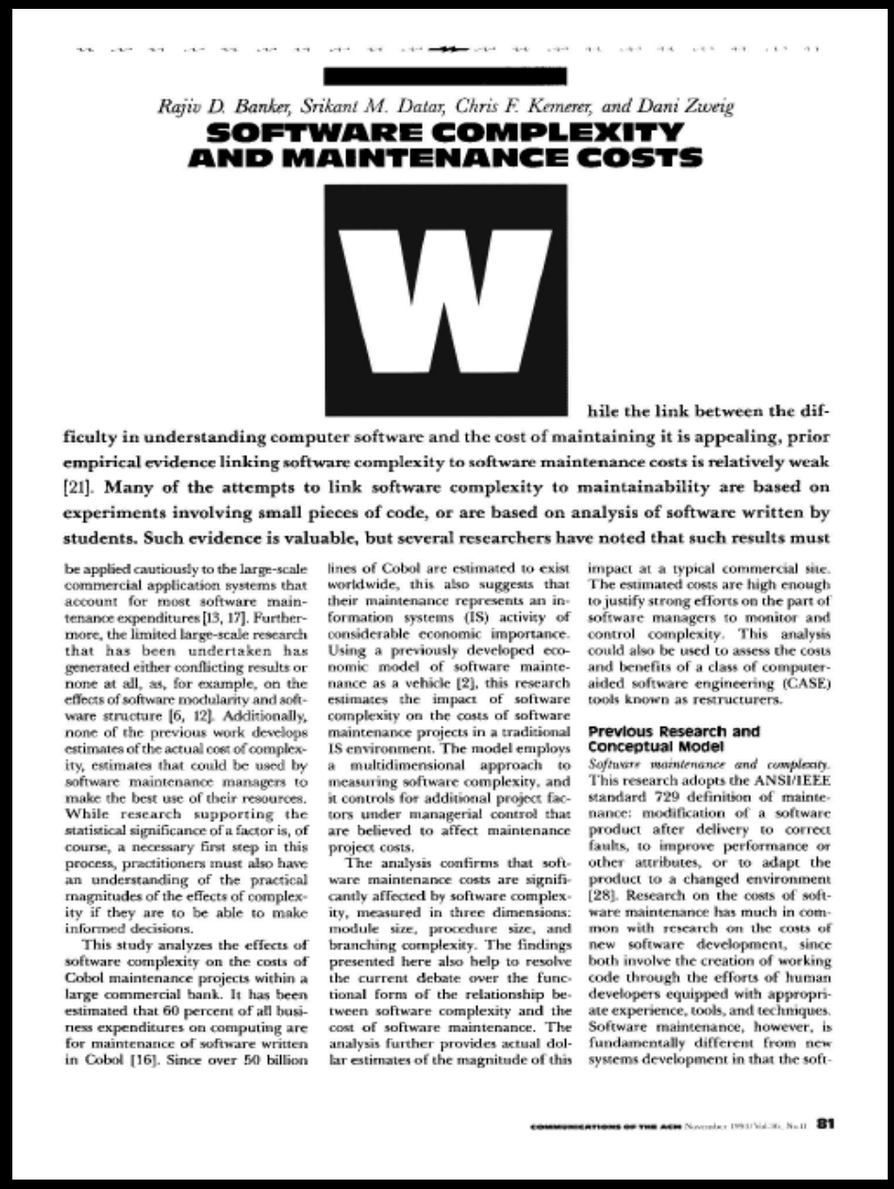
Y.-G. Guéhéneuc
e-mail: yann-gael.gueheneuc@polymtl.ca

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

Bad Smells increase change- and fault-proneness

[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 be applied cautiously to the large-scale commercial application systems that account for most software maintenance expenditures [13, 17]. Furthermore, the limited large-scale research that has been undertaken has generated either conflicting results or none at all, as, for example, on the effects of software modularity and software structure [6, 12]. Additionally, none of the previous work develops estimates of the actual cost of complexity, estimates that could be used by software maintenance managers to make the best use of their resources. While research supporting the statistical significance of a factor is, of course, a necessary first step in this process, practitioners must also have an understanding of the practical magnitudes of the effects of complexity if they are to be able to make informed decisions.

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

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

Previous Research and Conceptual Model

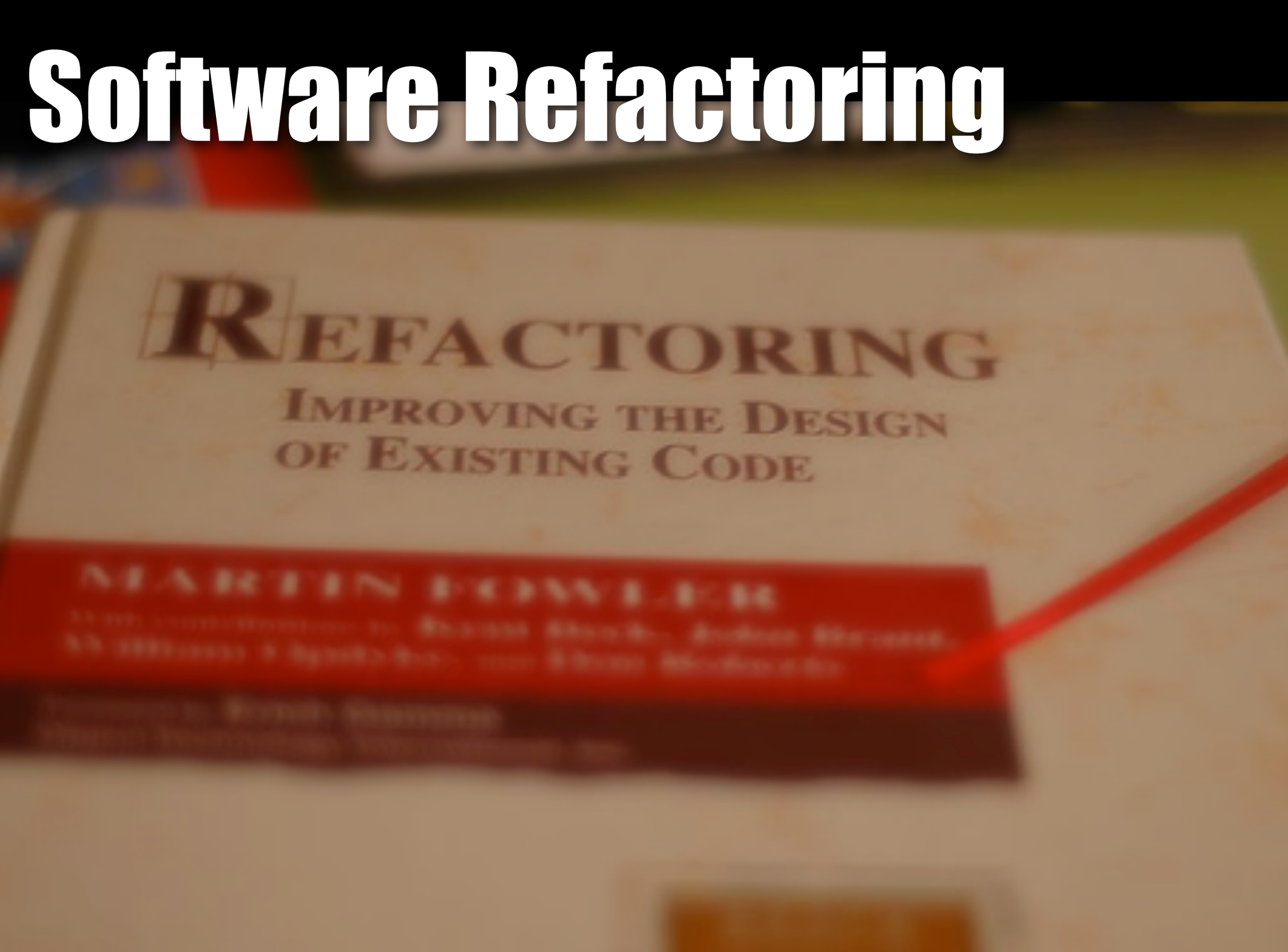
Software maintenance and complexity. This research adopts the ANSI/IEEE standard 729 definition of maintenance: modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment [28]. Research on the costs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

Bad Smells increase maintenance costs

[Banker et al. Communications of the ACM]

How do we cope with Bad Code Smells ?

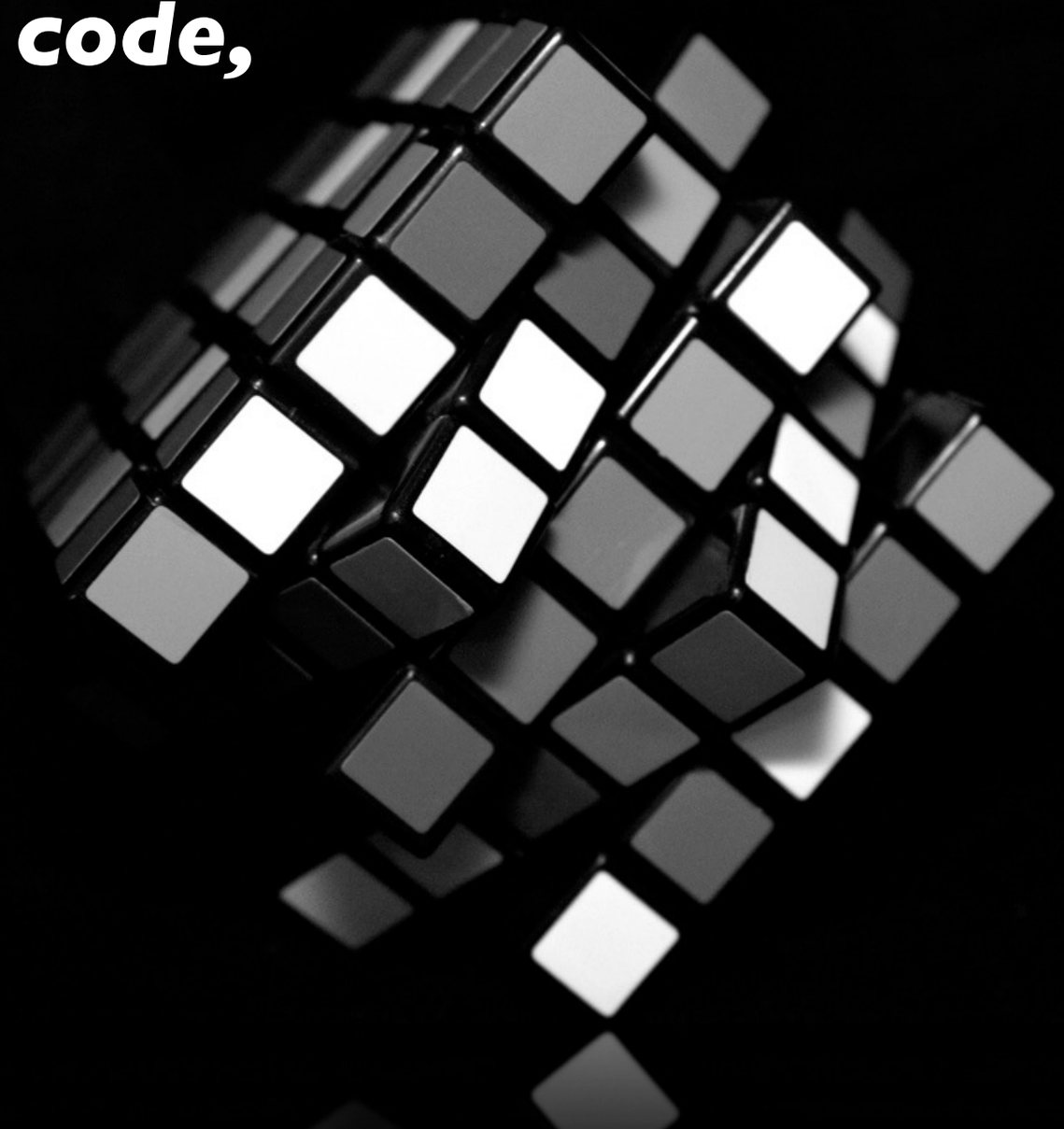
Software Refactoring



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

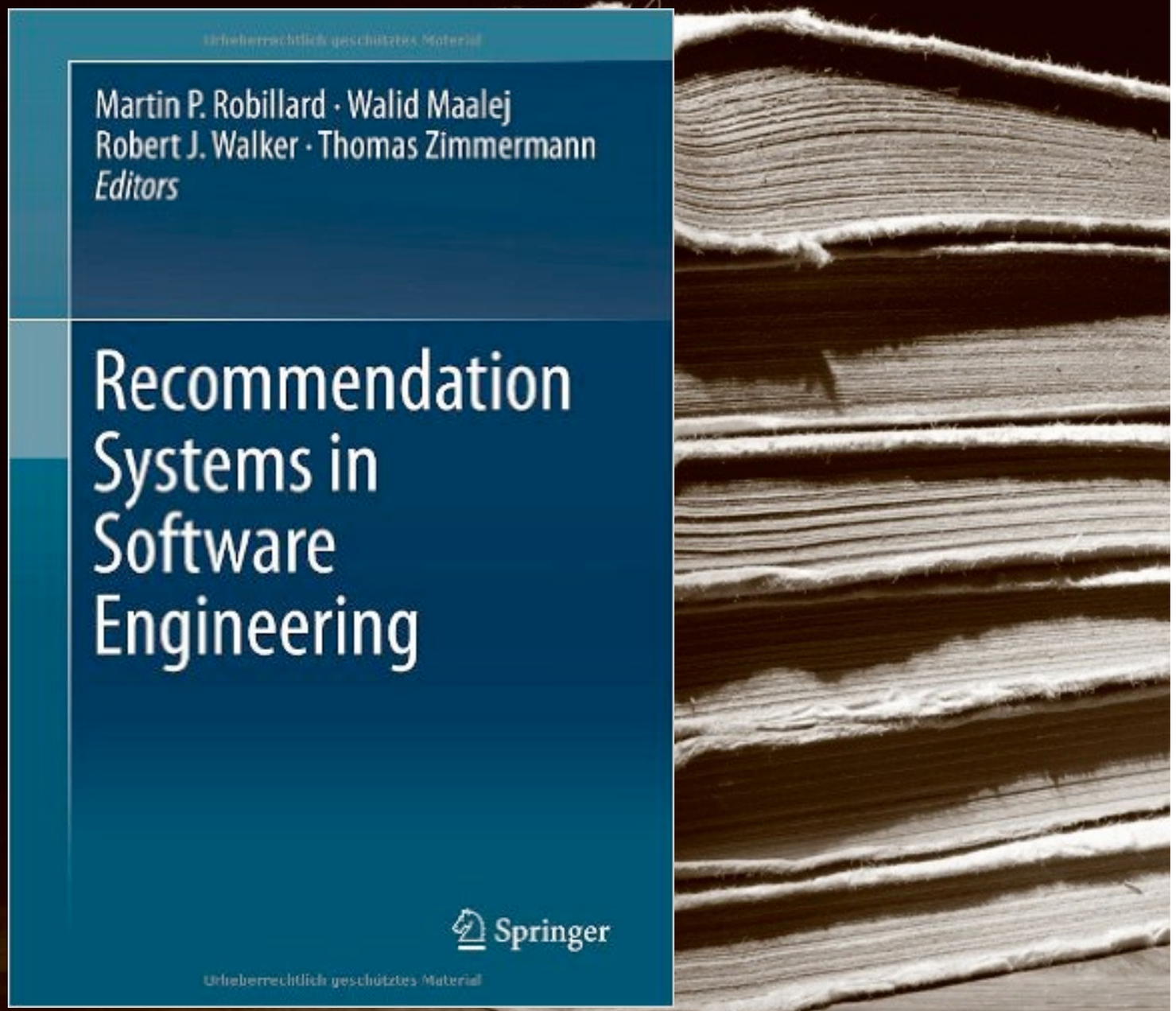
add paremeter

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

Cohesion

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

Cohesion

Cohesion

High cohesion is desirable

Coupling

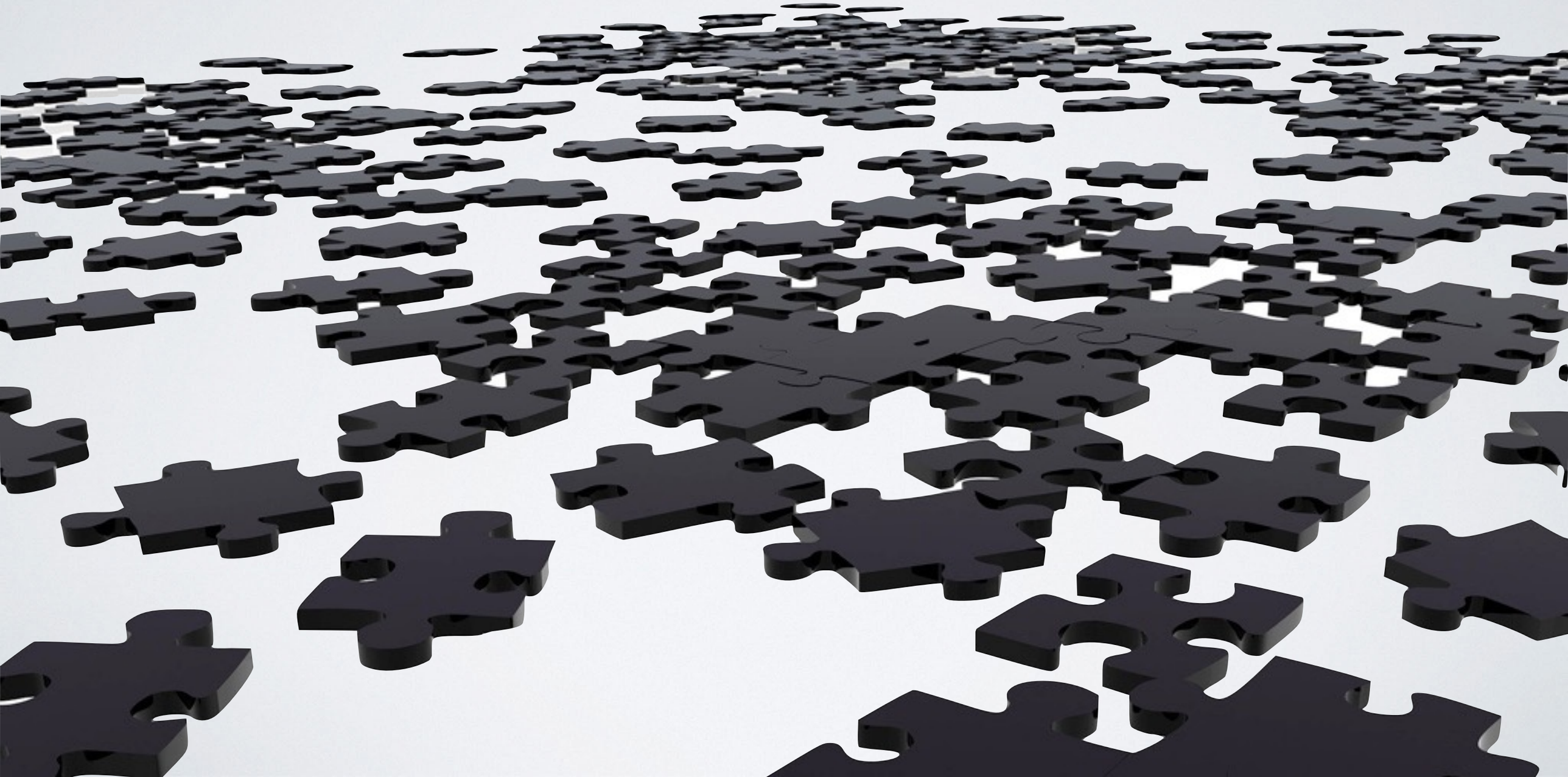
Coupling

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

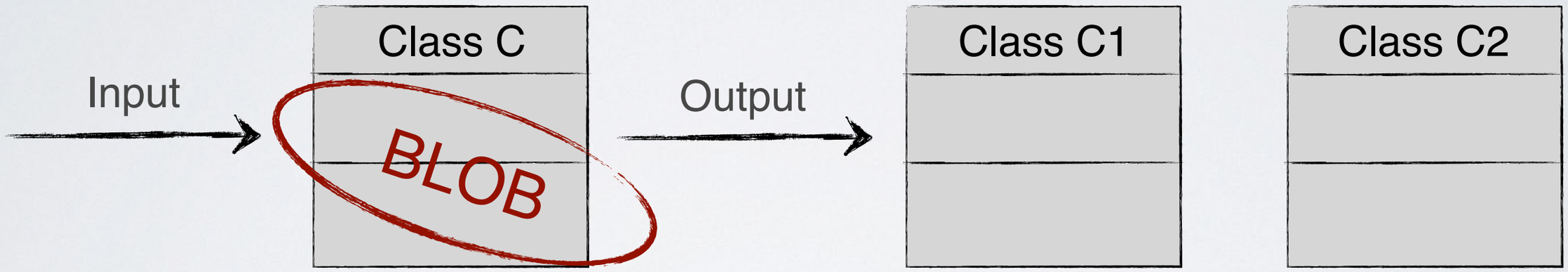
Coupling

Coupling

Low coupling is desirable



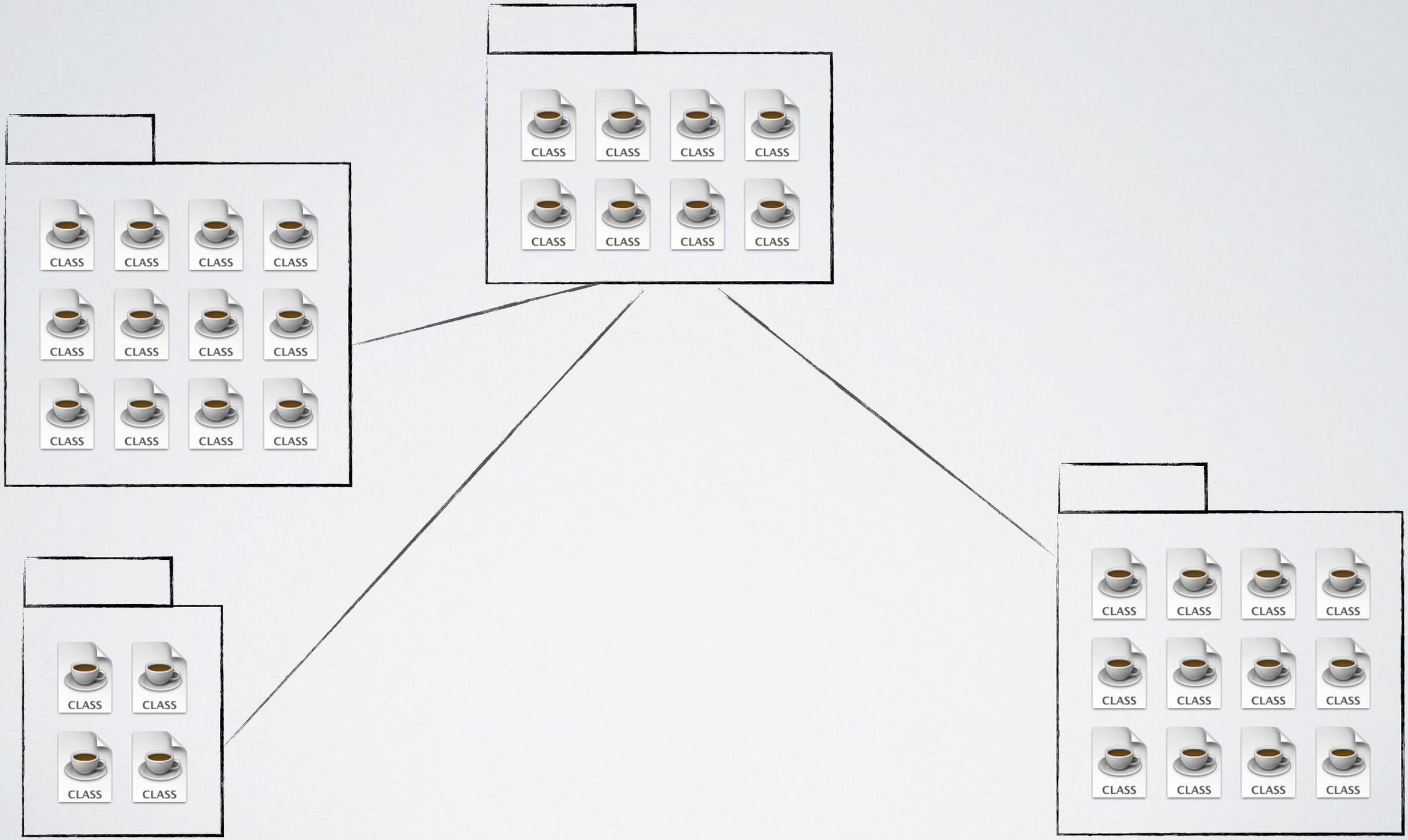
Extract Class Refactoring



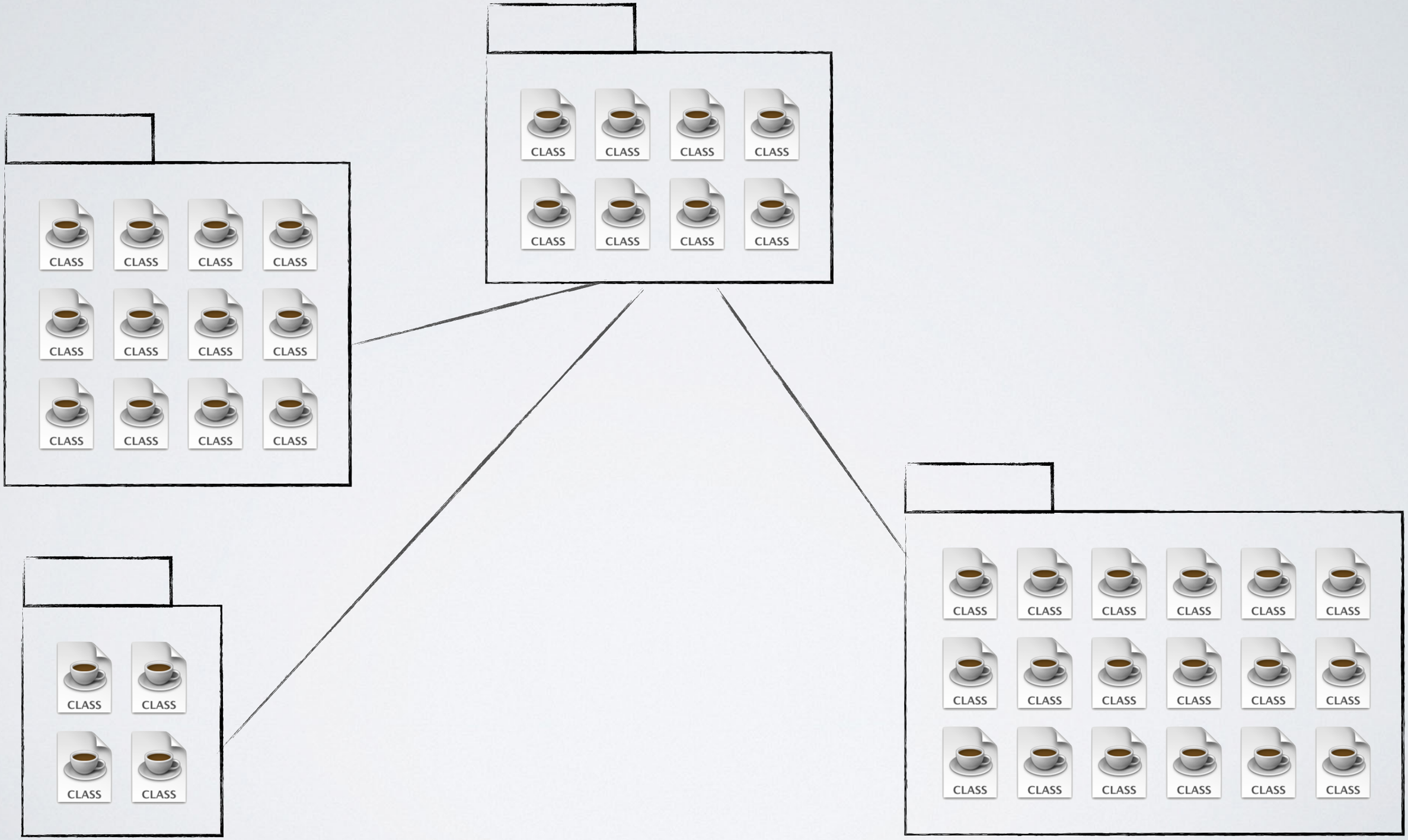
Splitting a class with many responsibilities into different classes

Extract Package Refactoring

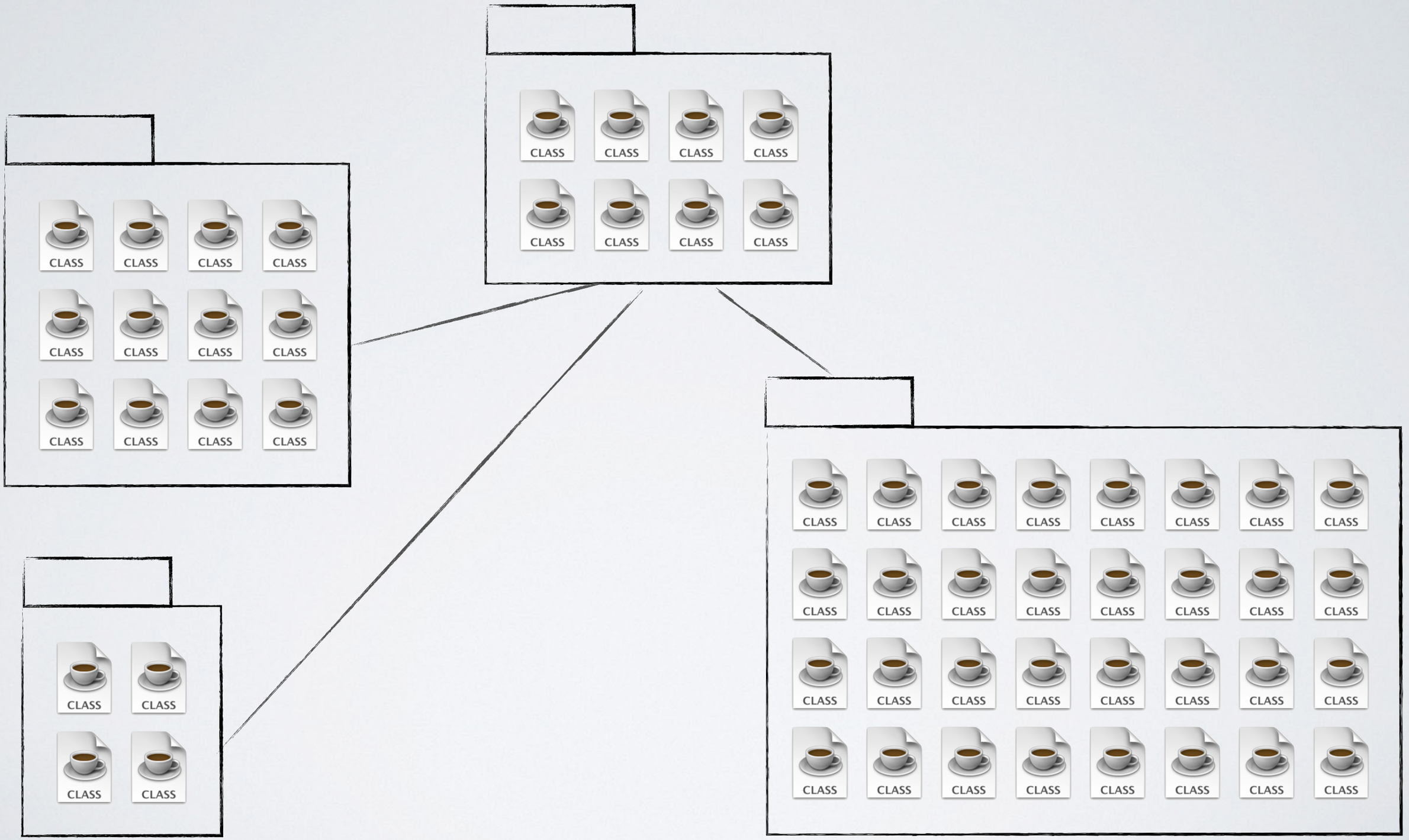
Promiscuous packages



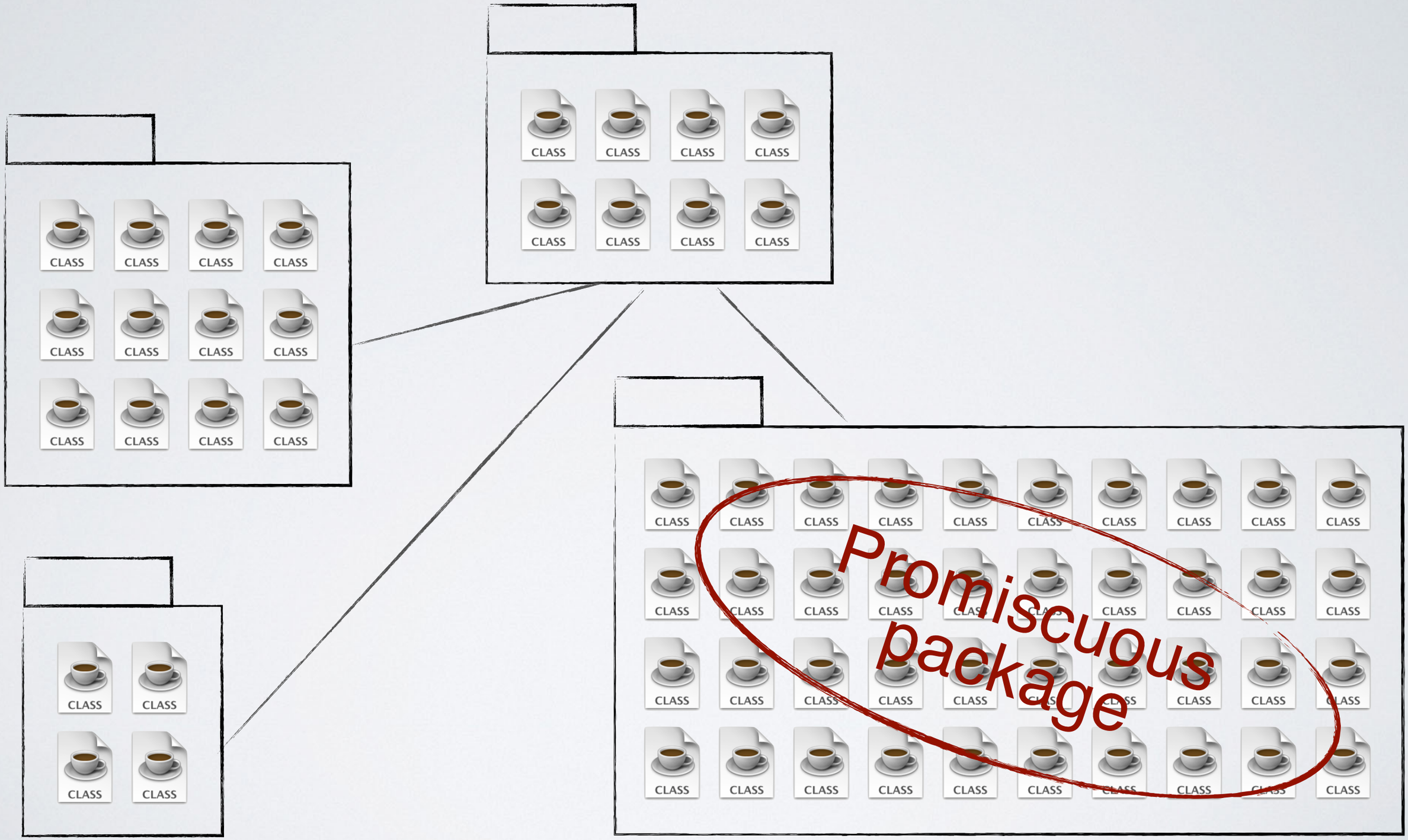
Promiscuous packages



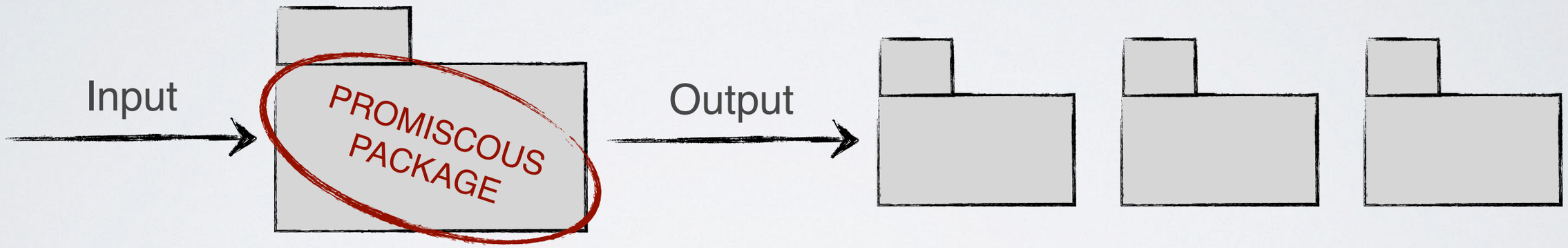
Promiscuous packages



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

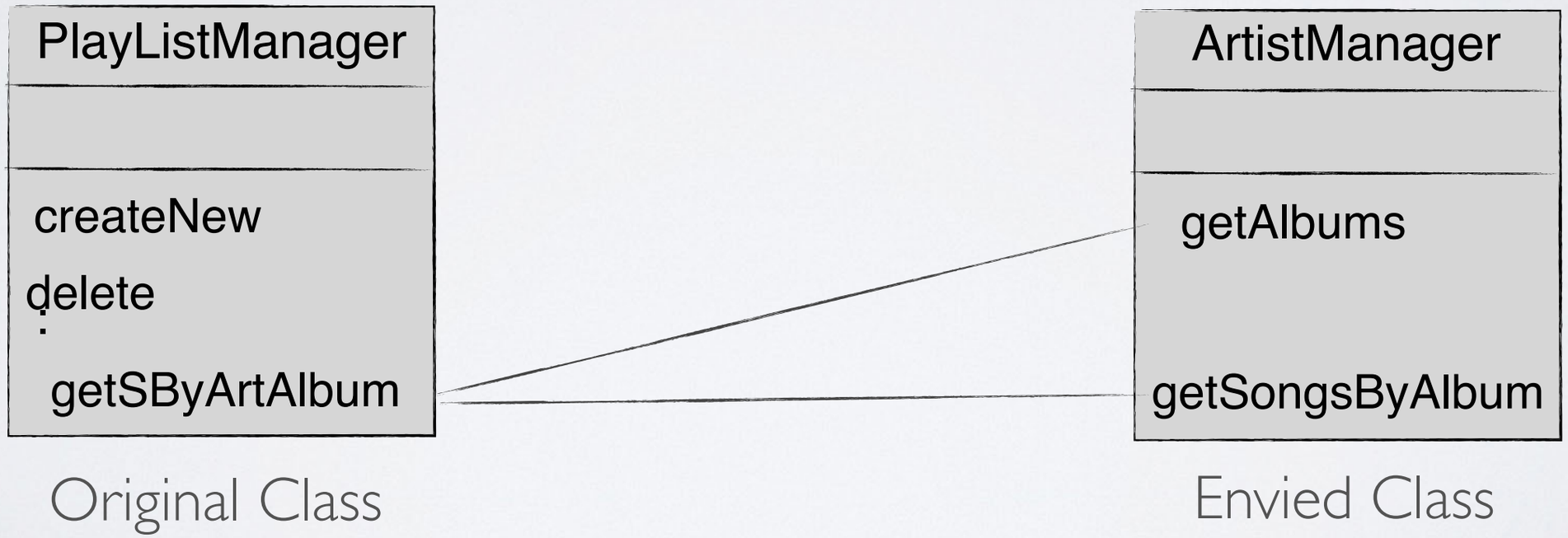
Feature Envy Bad Smell

**“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]

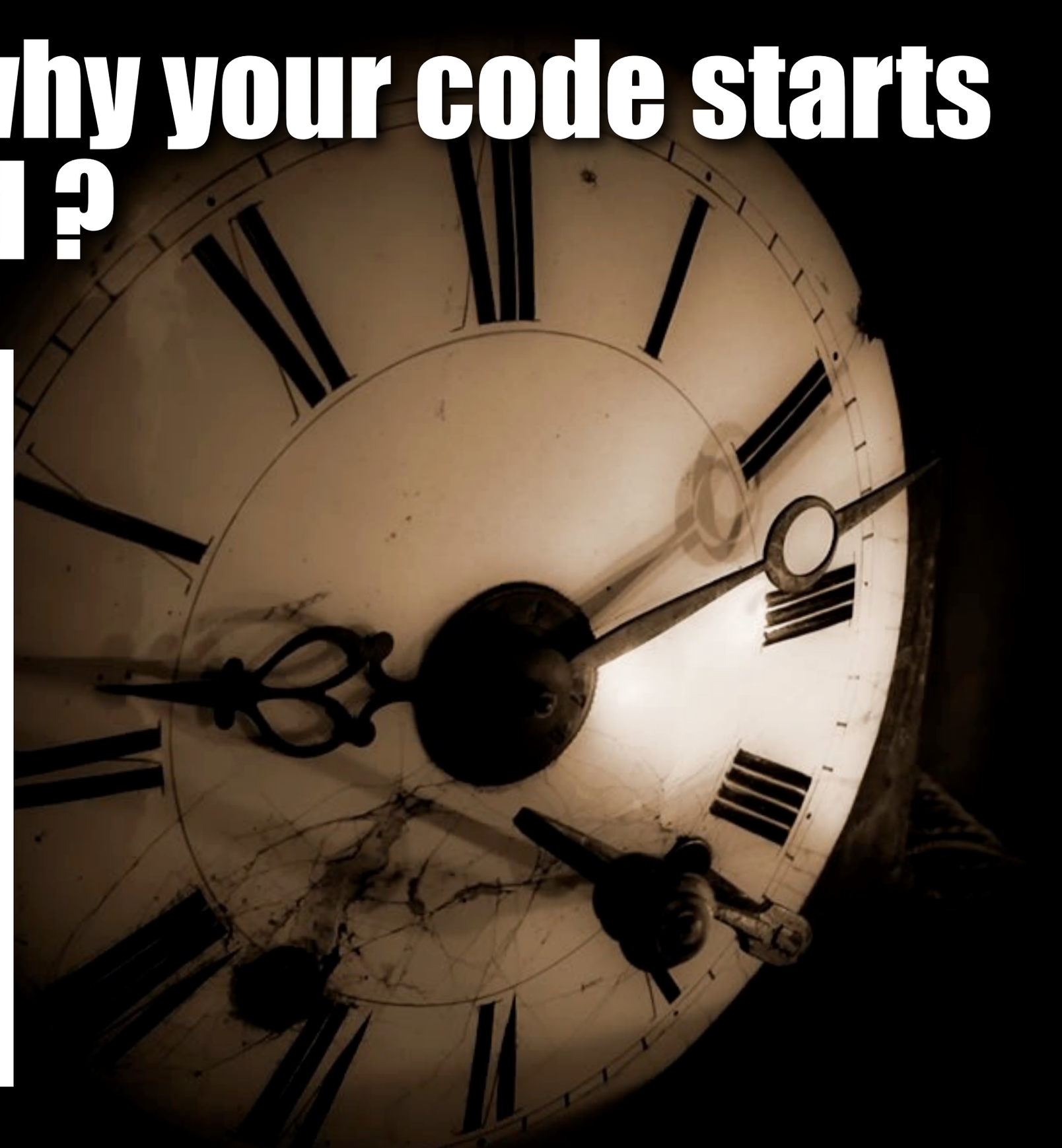
Fowler M.
Refactoring: improving the design of existing code
Addison-Wesley

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 ?



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

[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

ANDROID

eclipse

different ecosystems analyzed

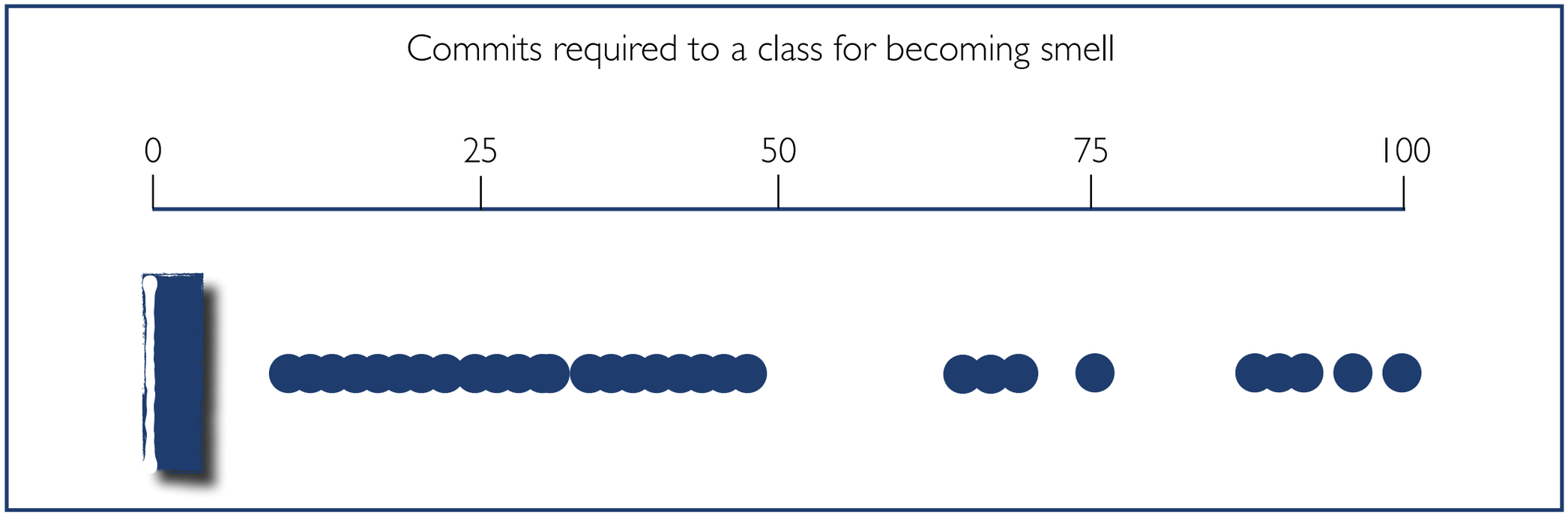
Study Design

200

total analyzed systems

When are code smells introduced

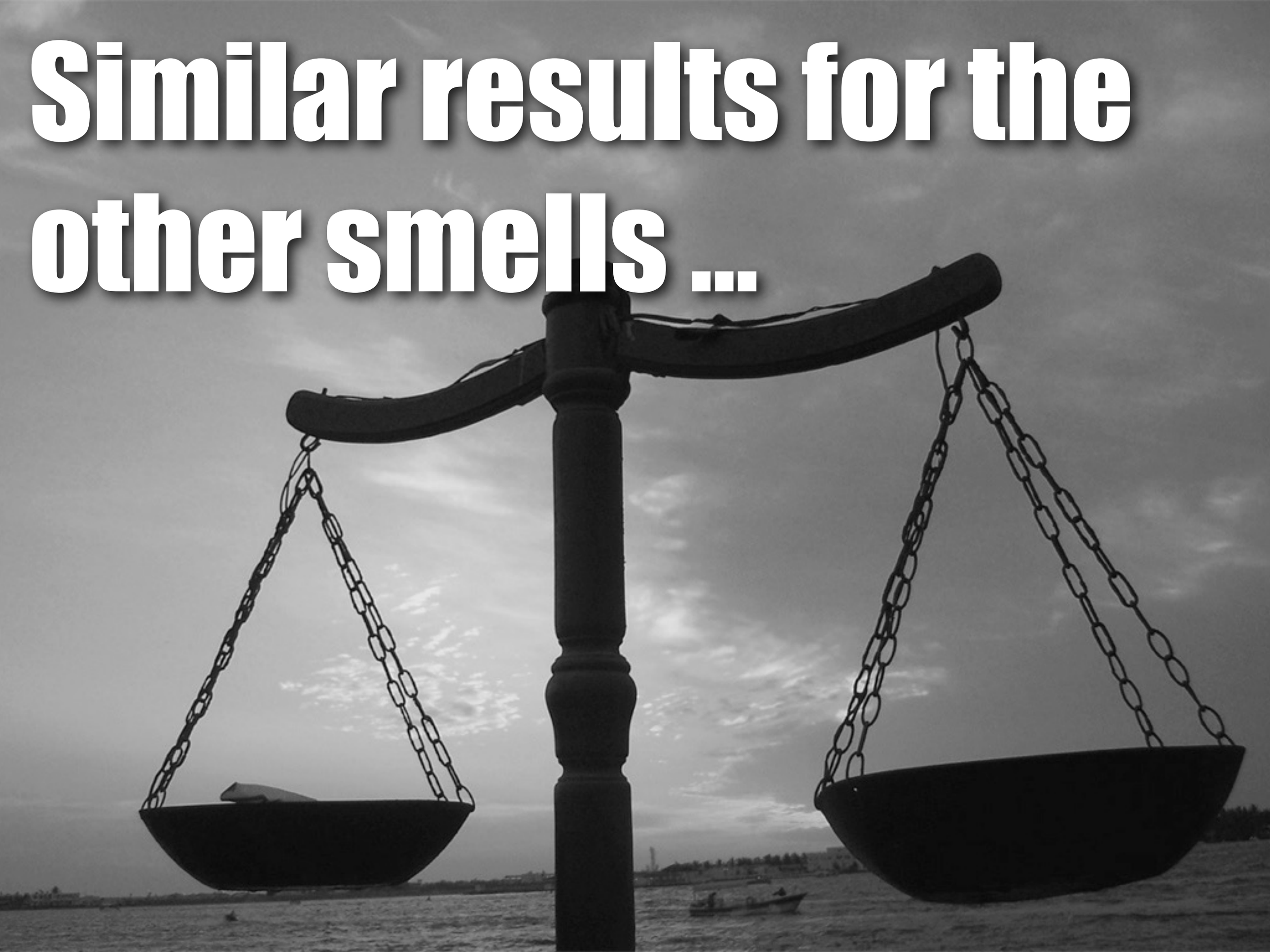
WHEN blobs are introduced

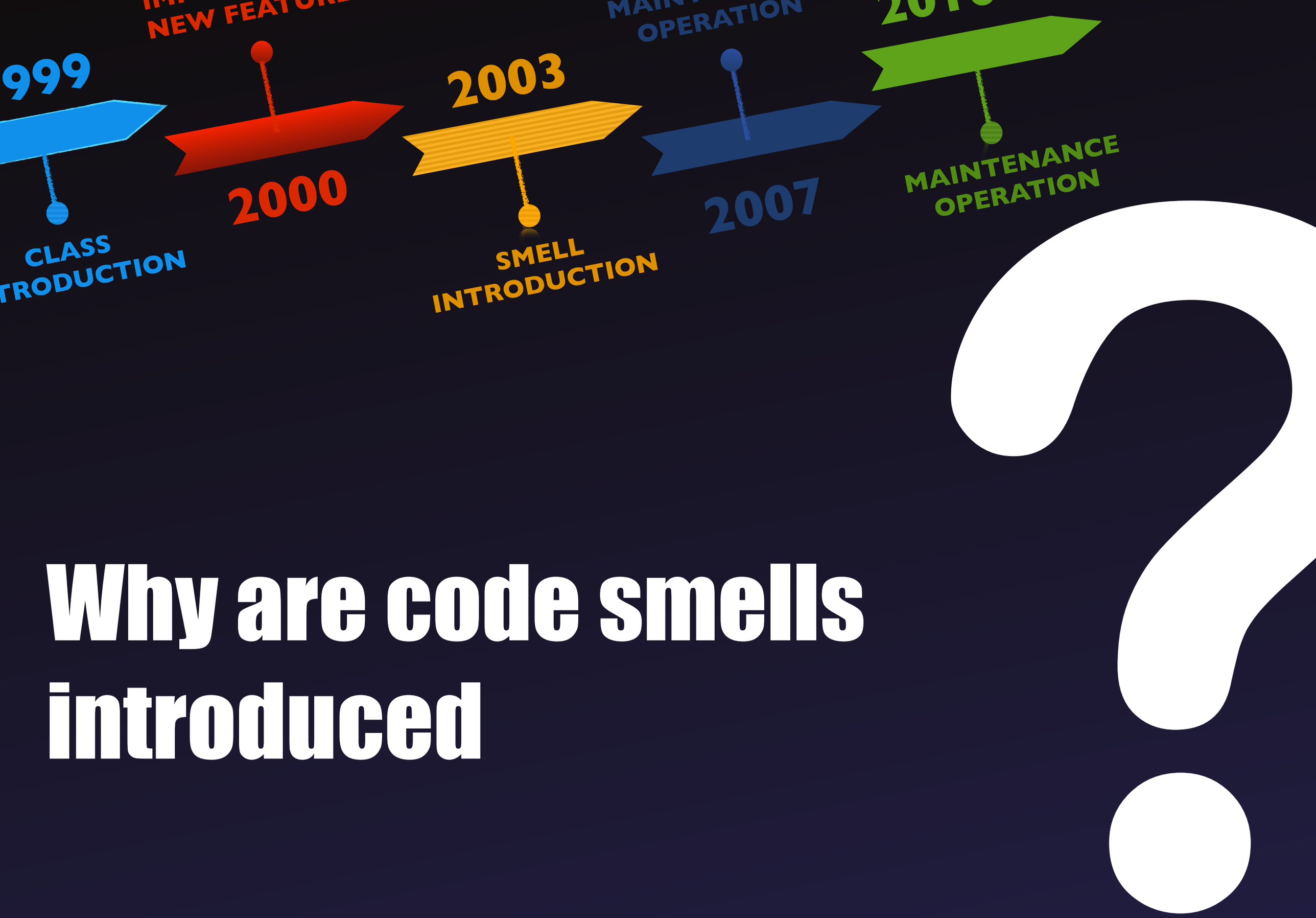


Generally, blobs affect a class since its creation

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

Similar results for the other smells ...





Why are code smells introduced

WHY are code smells introduced

Maintenance Activity

Bug Fixing

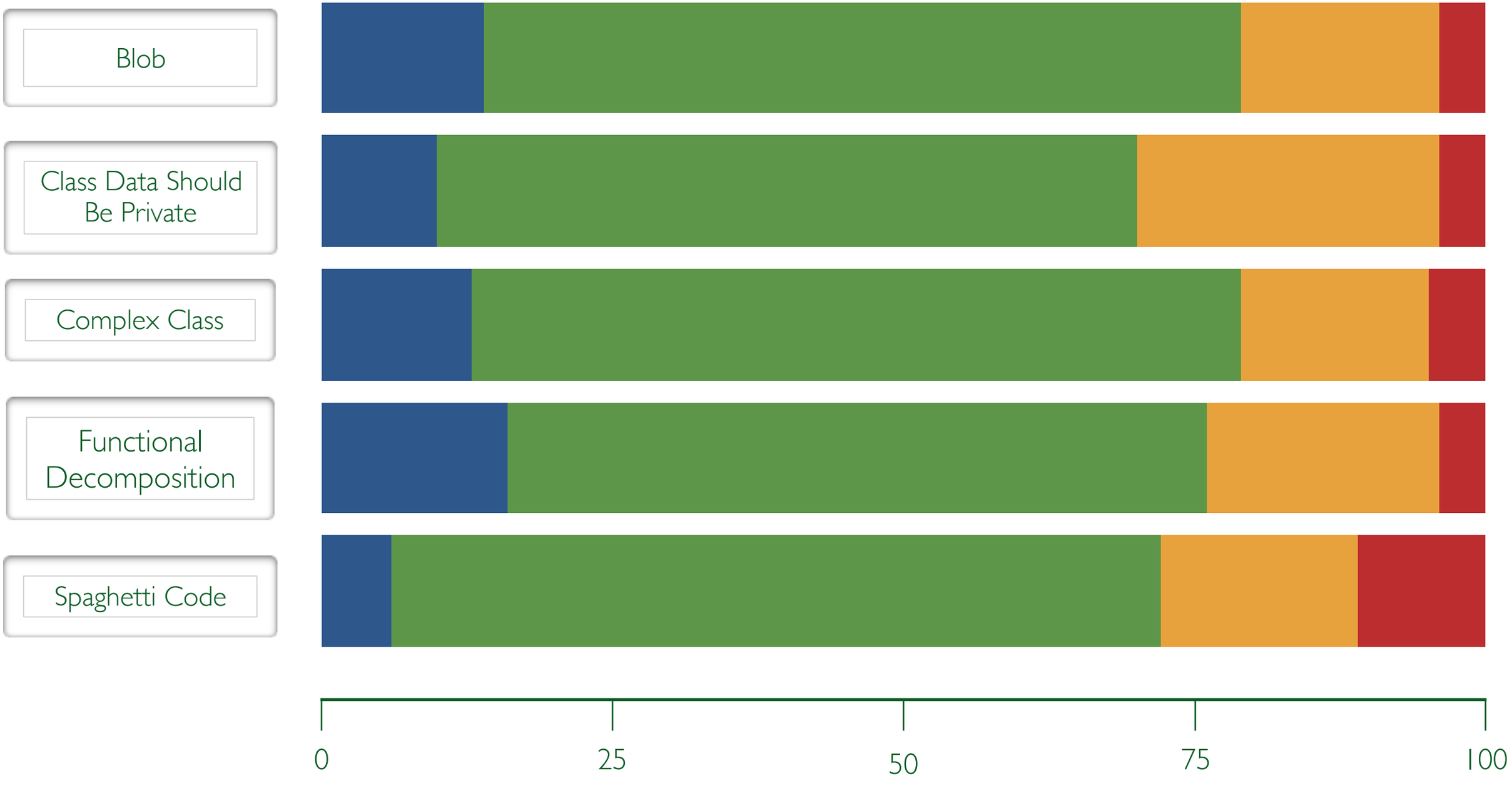
Enhancement

New Feature

Refactoring

WHY are code smells introduced

Maintenance Activity



WHY are code smells introduced

Workload

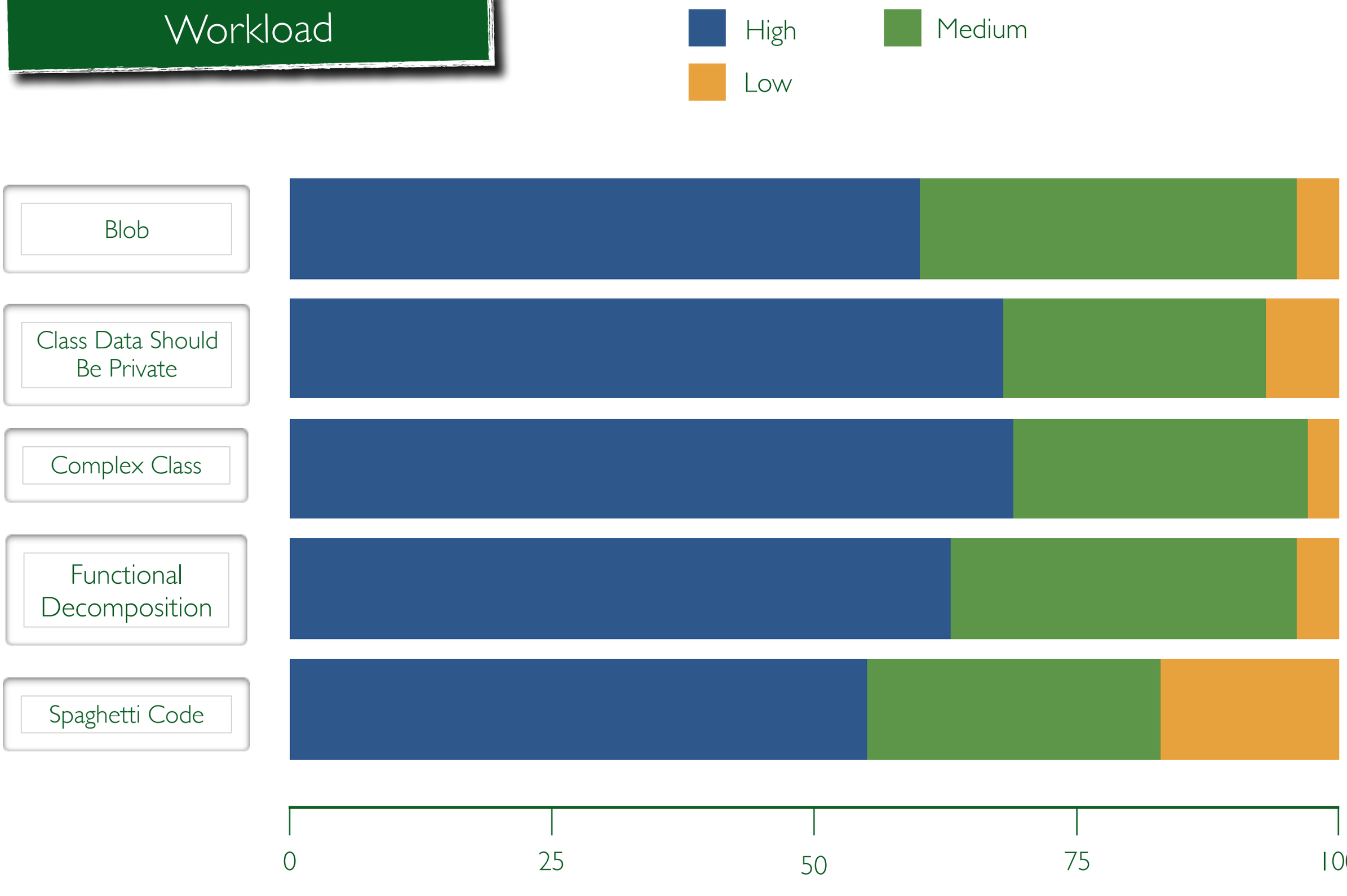
High

Medium

Low

WHY are code smells introduced

Workload



WHY are code smells introduced

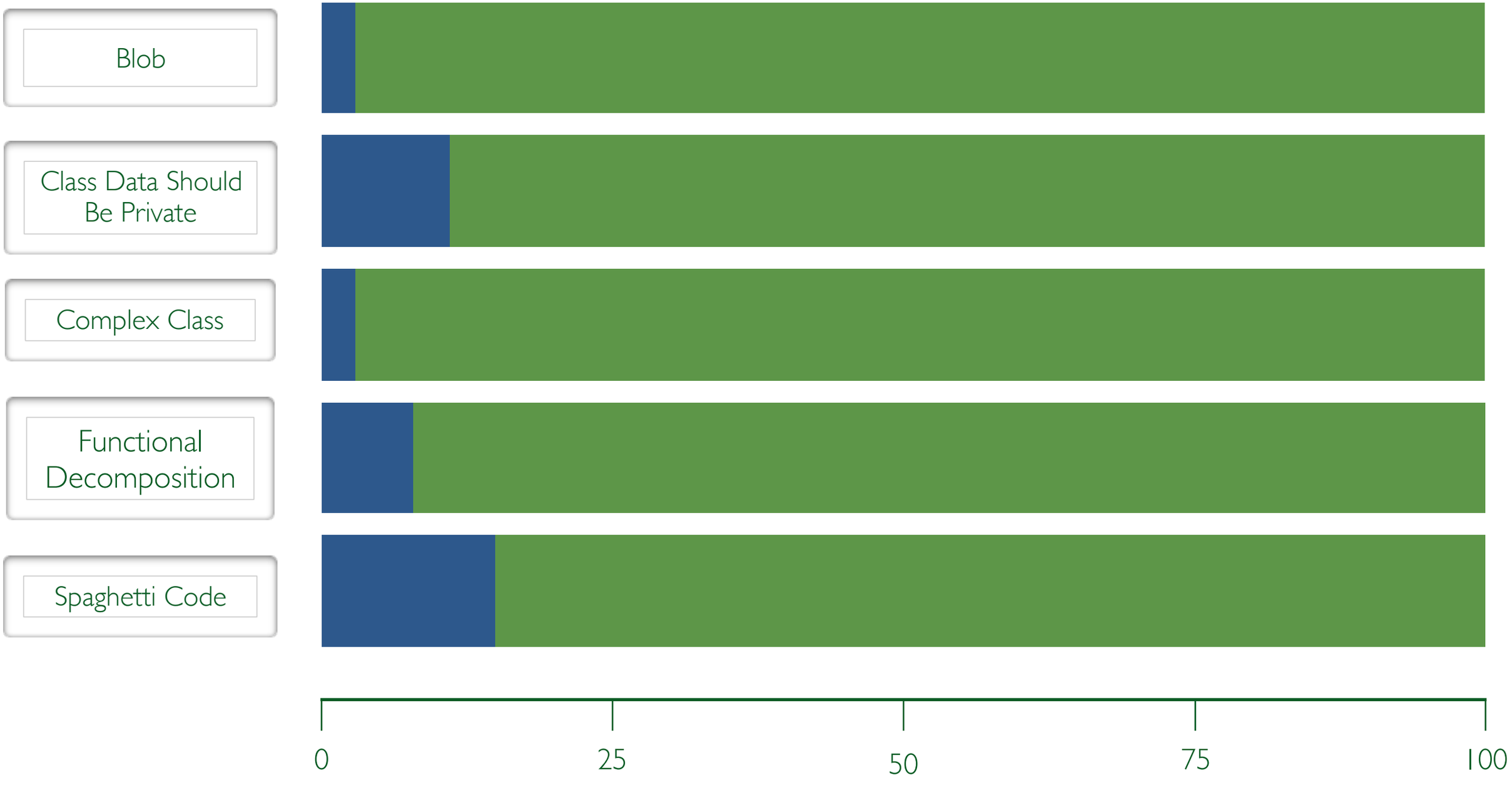
Newcomer

True

False

WHY are code smells introduced

Newcomer



How about the Developers' Perception of Bad Code Smells ?

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

Do Developers Care About Code Smells? – An Exploratory Survey

Aiko Yamashita
Mesan 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)

daffy.it

Do Developers Care About Code Smells? – An Exploratory Survey

Aiko Yamashita
Mesan 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

Yamashita and Moonen (WCRE 2013)

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

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[2], Rocco Oliveto[3], Andrea De Lucia[1]
[1]University of Salerno, Italy [2]University of Sannio, Italy [3]University of Molise, Italy

Abstract—In the last decade several catalogues have been defined to characterize bad code smells, i.e., symptoms of poor design and implementation choices. On top of such catalogues, researchers have defined methods and tools to automatically detect and/or remove bad smells. Nevertheless, there is an ongoing debate regarding the extent to which developers perceive bad smells as serious design problems. Indeed, there seems to be a gap between theory and practice, i.e., what is believed to be a problem (theory) and what is actually a problem (practice). This paper presents a study aimed at providing empirical evidence on how developers perceive bad smells. In this study, we showed to developers code entities—belonging to three systems—affected and not by bad smells, and we asked them to indicate whether the code contains a potential design problem, and if any, the nature and severity of the problem. The study involved both original developers from the three projects and outsiders, namely industrial developers and Master's students. The results provide insights on characteristics of bad smells not yet explored sufficiently. Also, our findings could guide future research on approaches for the detection and removal of bad smells.

Index Terms—Code Smells, Empirical Study

I. Introduction

Bad code smells represent symptoms of poor design and implementation choices [1]. Bad smells are usually introduced in software systems because developers poorly conceived the design of the code component or because they did not care about properly designing the solution due to strict deadlines. *Complex Class*, i.e., a class that contain complex methods and it is very large in terms of LOC; or *God Class*, i.e., a class that does too much/knows too much about other classes, are only some examples of a plethora of bad smells identified and characterized in well-known catalogues [1], [2].

Recent empirical studies showed that code smells hinder comprehensibility [3], and possibly increase change- and fault-proneness [4]. Also, the interaction between different, co-existing code smells can negatively affect maintainability [5]. Hence, there is empirical evidence that code smells have a negative effect on software evolution, and therefore should be carefully monitored and possibly removed through refactoring operations. Thus, a lot of effort has been devoted for the definition of approaches aiming at detecting and removing bad code smells [6], [7], [8], [9].

Despite the existing evidence about the negative effects of code smells [3], [4], [5], and the effort devoted to the definition of approaches for detecting and removing them, it is still unclear whether developers would actually consider all bad smells as actual symptoms of wrong design/implementation choices, or whether some of them are simply a manifestation of the intrinsic complexity of the designed solution. In other words, there seems to be a gap between the theory and the practice. For example, a recent study found that some source code files of the Linux Kernel intrinsically have high cyclomatic complexity. However, this is not considered a design or implementation problem by developers [10]. Also, empirical studies indicated that (i) God Classes sporadically changing are not felt as a problem by developers [11]; and (ii) some developers, in particular junior programmers, work better on a version of a system having some classes that centralized the control, i.e., God classes [12]. These results suggest that the presence of bad smells in source code is sometimes tolerable, and part of developers' design choices.

Recently, Yamashita and Moonen [13] performed an exploratory survey aimed at investigating developers knowledge about code smells, by asking questions like "How familiar are you with bad code smells?". Results showed that a large proportion of respondents did not know about bad code smells. While the study of Yamashita and Moonen aimed at investigating to what extent developers had a theoretical knowledge of code smells (i.e., knowing them from their name and definition), no study so far investigated whether, given a problem instance—that can be brought back to the presence of a bad smell in the code—developers actually perceive the problem as such and whether they associate the problem to the same symptoms explained in the smell definition.

To bridge this gap, we conducted a study aimed at investigating the developers' perception of code smells. First, we identified and manually validated instances of 12 different bad smells in three open source projects. Then, we provided a questionnaire to the participants where we showed code snippets affected and not affected by bad smells, and asked whether, in the respondents' opinion, the code component has any problem. In case of a positive answer, we asked them to explain what kind of problem they perceived and how severe they judged it. We asked different categories of subjects to participate in the study, namely (i) Master's students, representing a population of subjects pretty knowledgeable about the theoretical concepts of code smells, (ii) industrial developers, i.e., people with experience on real development projects, but not knowing the code being shown; and (iii) developers from the open-source projects in which the bad smells have been collected. In total, we received responses from 34 subjects, and specifically 15 Master's students, 9 industrial developers, and 10 original developers of the studied projects. The data used in our study are publicly available as

1063-6773/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSME.2014.32

IEEE computer society

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

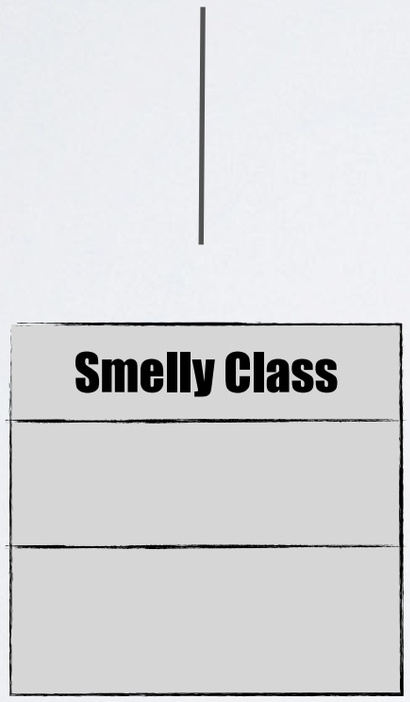
A class inheriting functionalities that it never uses

Speculative Generality

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

Study Design

Developer



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

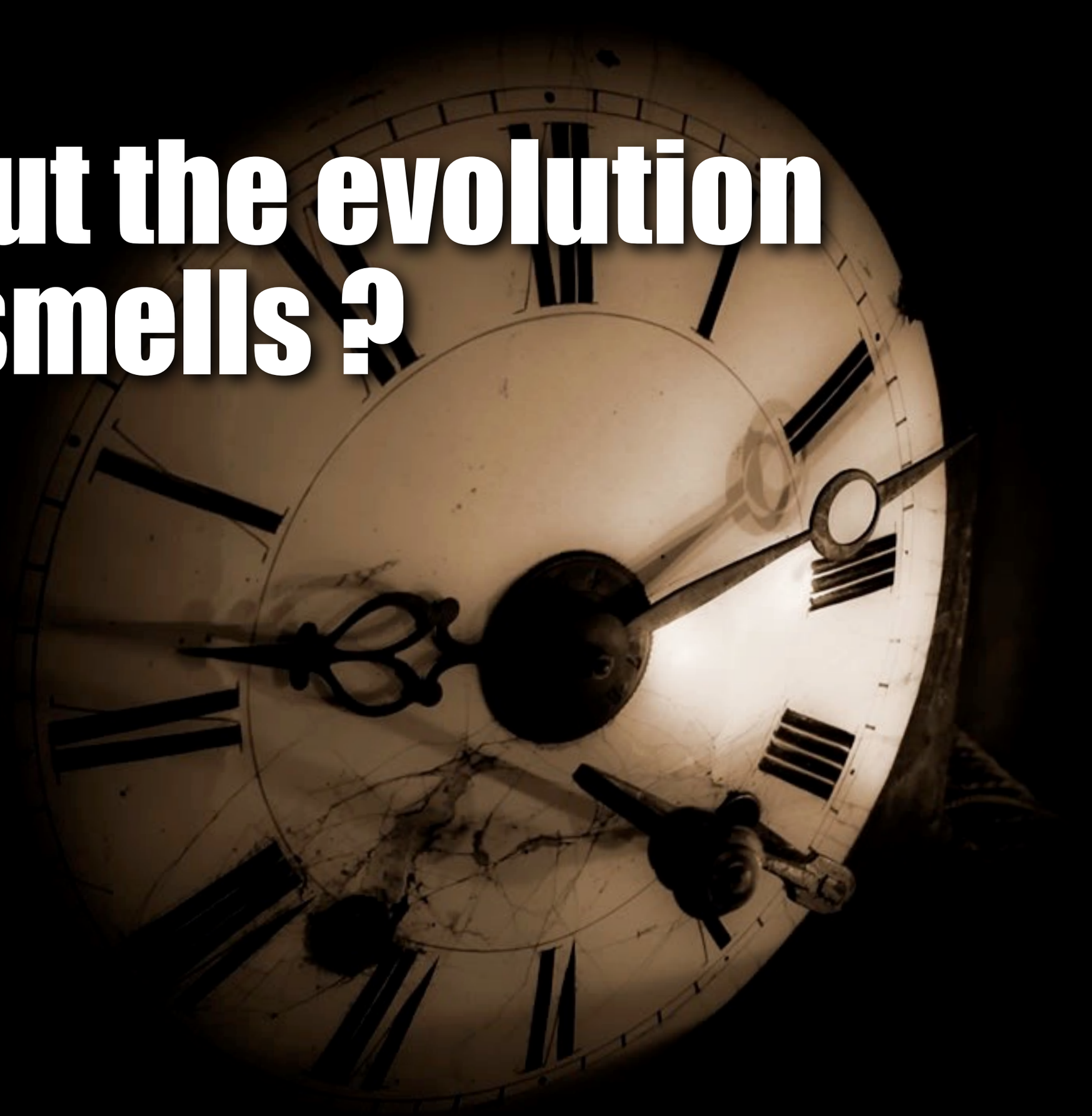
Refused Bequest

Feature Envy

Speculative Generality

DEPENDS ON THE "SEVERITY" OF THE PROBLEM

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: ralphpeters85@gmail.com

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

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

I. Introduction

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

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

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

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

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

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

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

II. Background

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

A. Code Smells

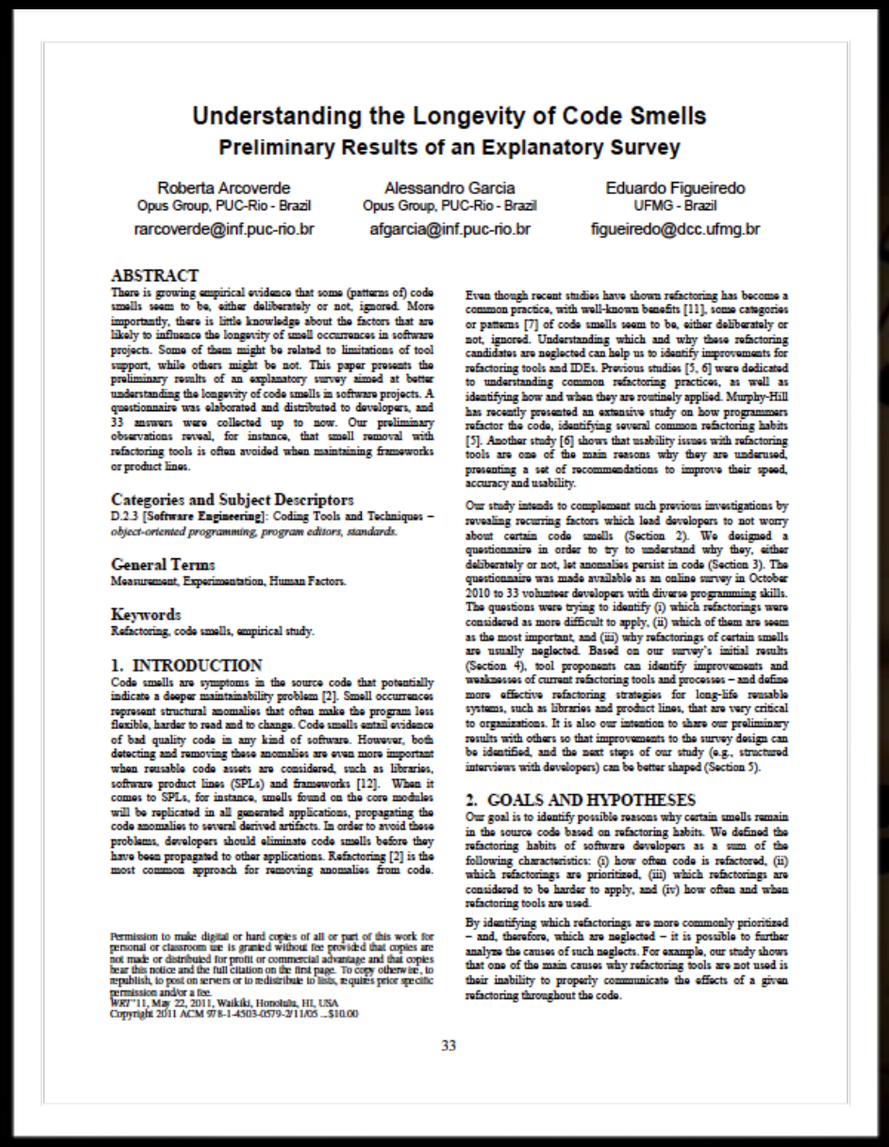
There is no widely accepted definition of code smells. In the introduction, we described code smells as symptoms of a deeper problem, also known as an anti-pattern. In fact, code smells can be considered anti-patterns at programming level rather than design level. Smells such as large classes and methods,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

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

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

Categories and Subject Descriptors

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

General Terms

Measurement, Experimentation, Human Factors.

Keywords

Refactoring, code smells, empirical study.

1. INTRODUCTION

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

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

Copyright 2011 ACM 978-1-4503-0579-2/11/05 ...$10.00

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

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

2. GOALS AND HYPOTHESES

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

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

Developers deliberately postpone refactorings for different reasons

[Arcoverde et al. - IWRT 2011]

... and their evolution

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

SI: QUATIC 2010

Investigating the evolution of code smells in object-oriented systems

Alexander Chatzigeorgiou · Anastasios Manakos

Received: 29 June 2011 / Accepted: 6 April 2013
© 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 in the history of a software project, we attempt to analyze the evolution from the point of view of the problems themselves. To this end, we classify smell evolution patterns distinguishing deliberate maintenance activities from the removal of design problems as a side effect of software evolution. Results are discussed for two open-source systems and four code smells.

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

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

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

Published online: 21 April 2013

Springer

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].

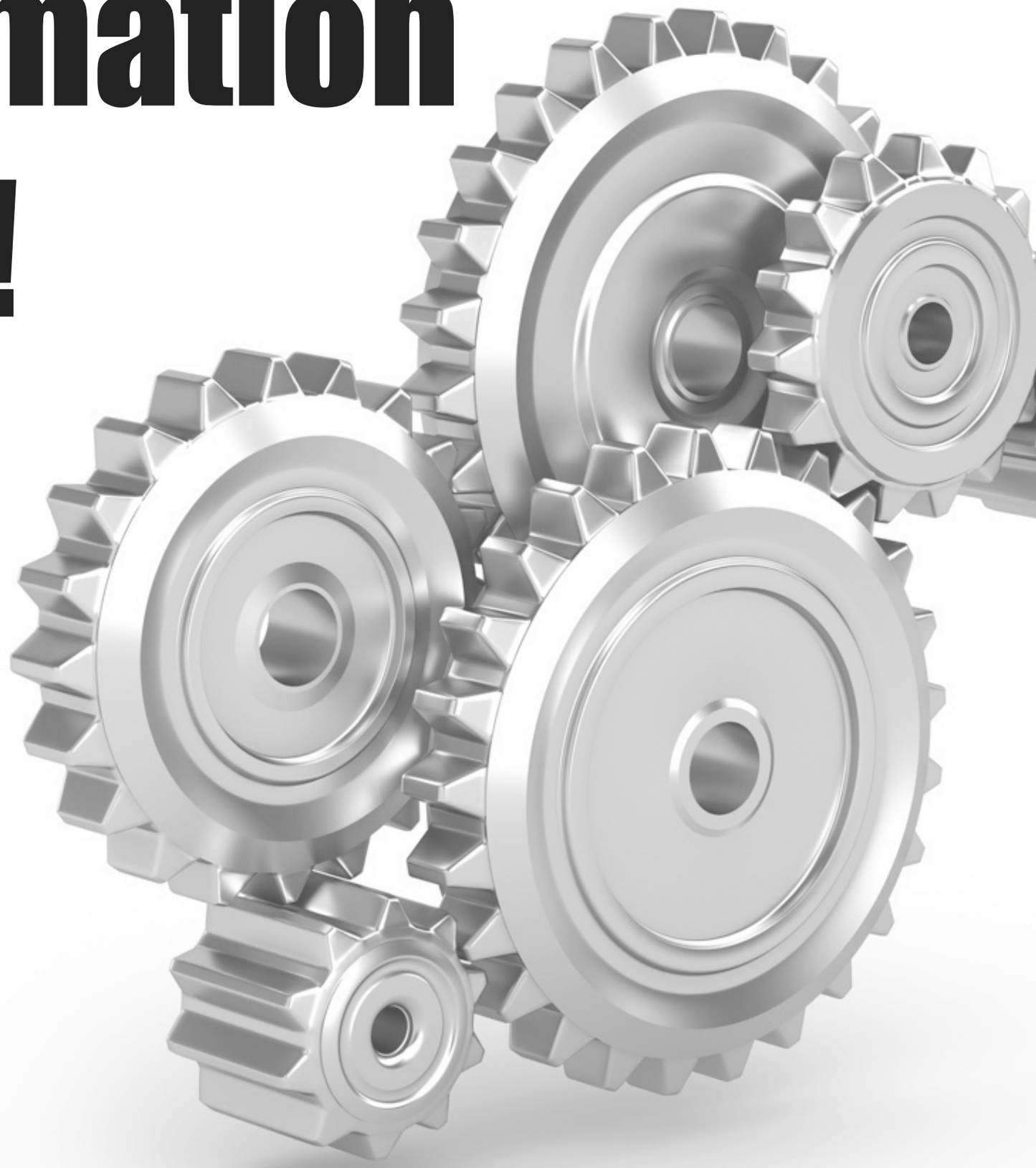
According to the recommendations proposed by Lehman and Ramil for software evolution planning [18], quality should be continuously monitored as systems evolve. This implies that past versions of a software system should be analyzed to track changes in evolutionary trends. To this end, organized collections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

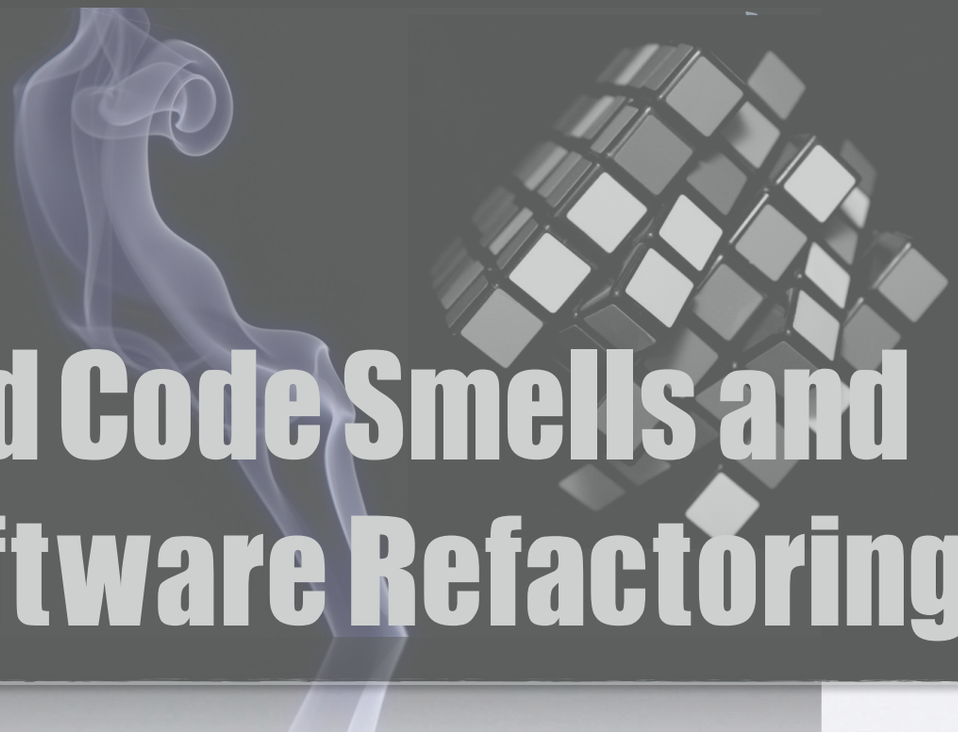
So what ?

More automation is needed !



both to identify and refactor bad smells

Part I

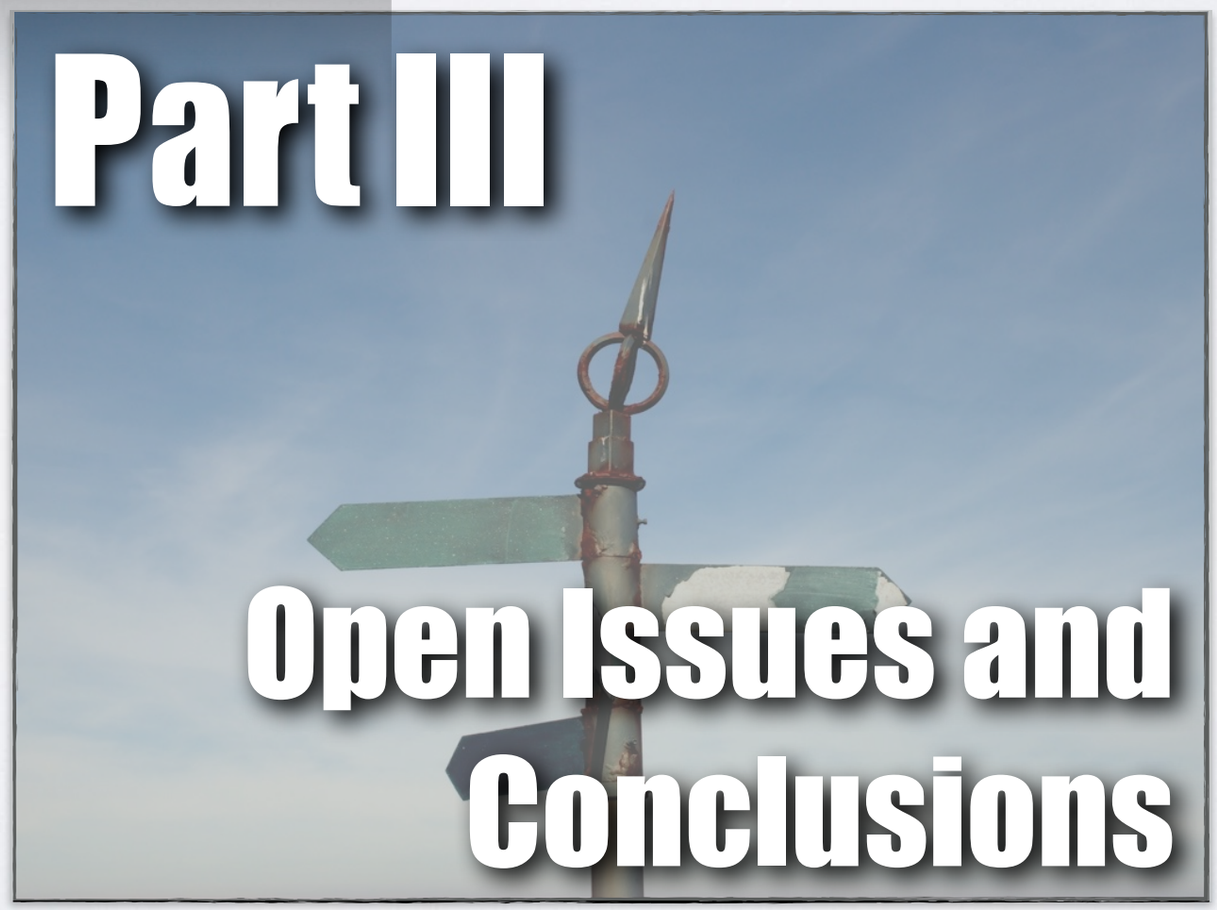


Bad Code Smells and Software Refactoring

Part II

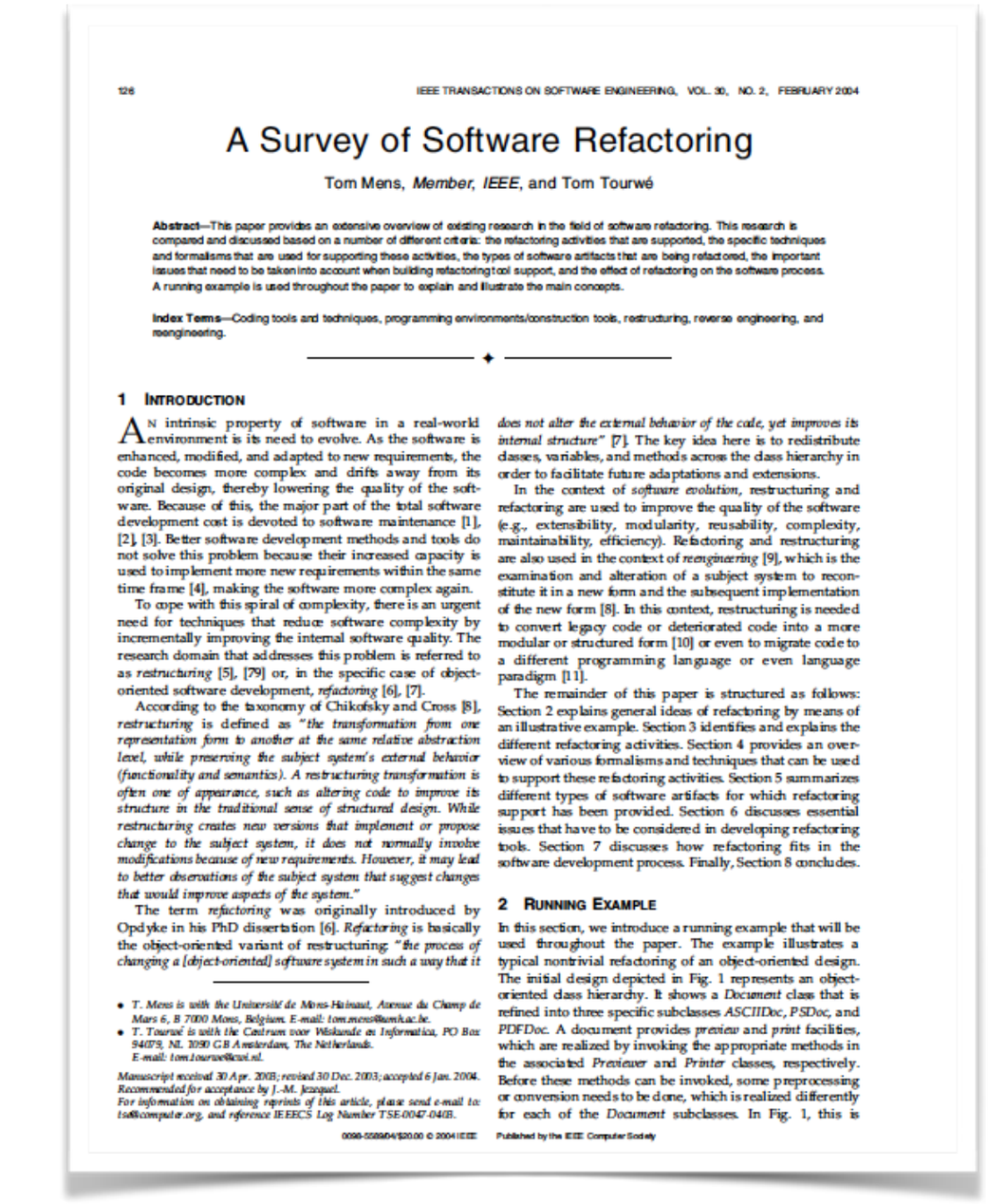
The refactoring process

Part III



Open Issues and Conclusions

The refactoring process



126 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 30, NO. 2, FEBRUARY 2004

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). Refactoring 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

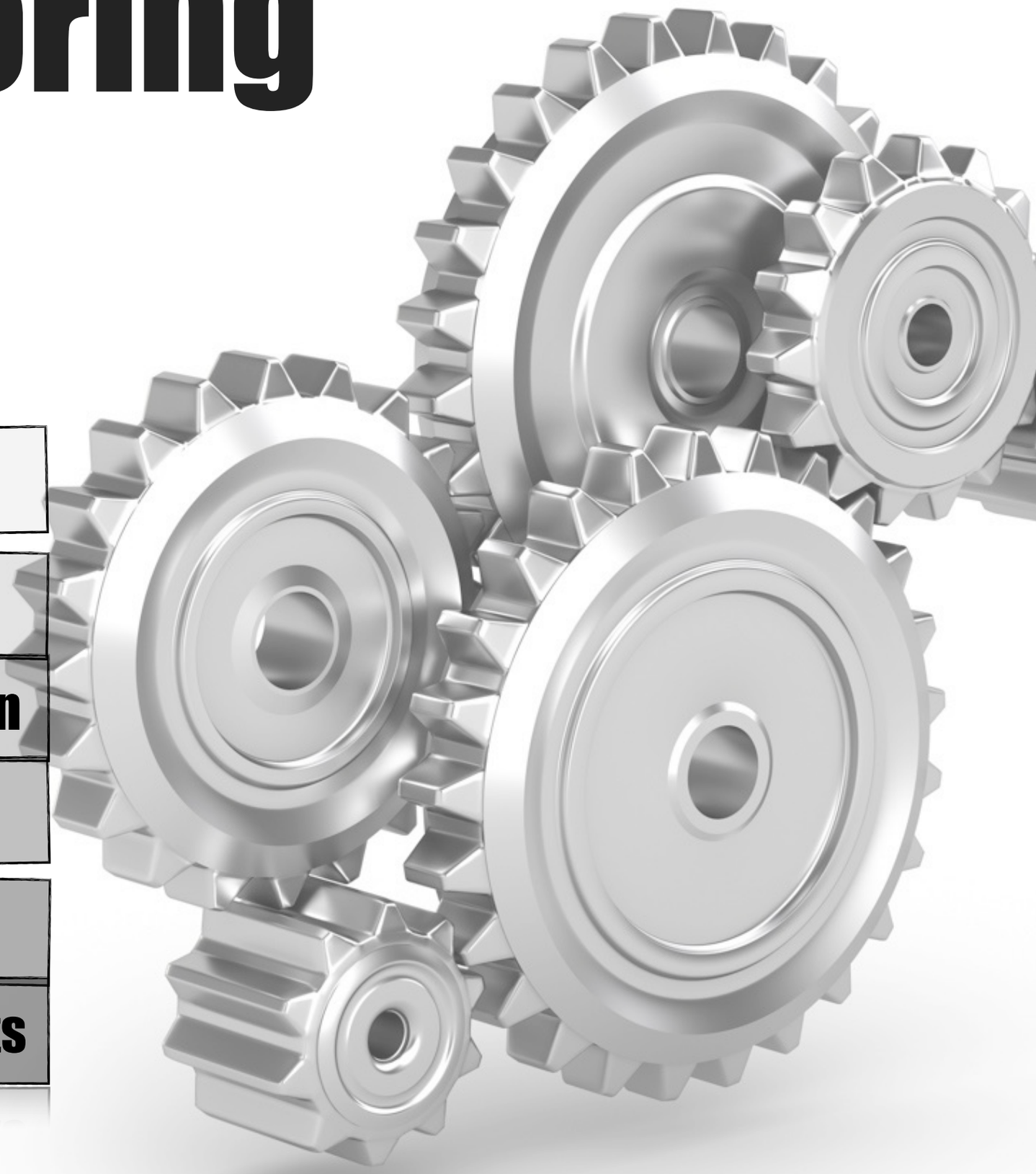
In this section, we introduce a running example that will be used throughout the paper. The example illustrates a typical nontrivial refactoring of an object-oriented design. The initial design depicted in Fig. 1 represents an object-oriented class hierarchy. It shows a *Document* class that is refined into three specific subclasses *ASCIIDoc*, *PSDoc*, and *PDFDoc*. A document provides *preview* and *print* facilities, which are realized by invoking the appropriate methods in the associated *Previewer* 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 30 Dec. 2003; accepted 6 Jan. 2004. Recommended for acceptance by J.-M. Jezequel.
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

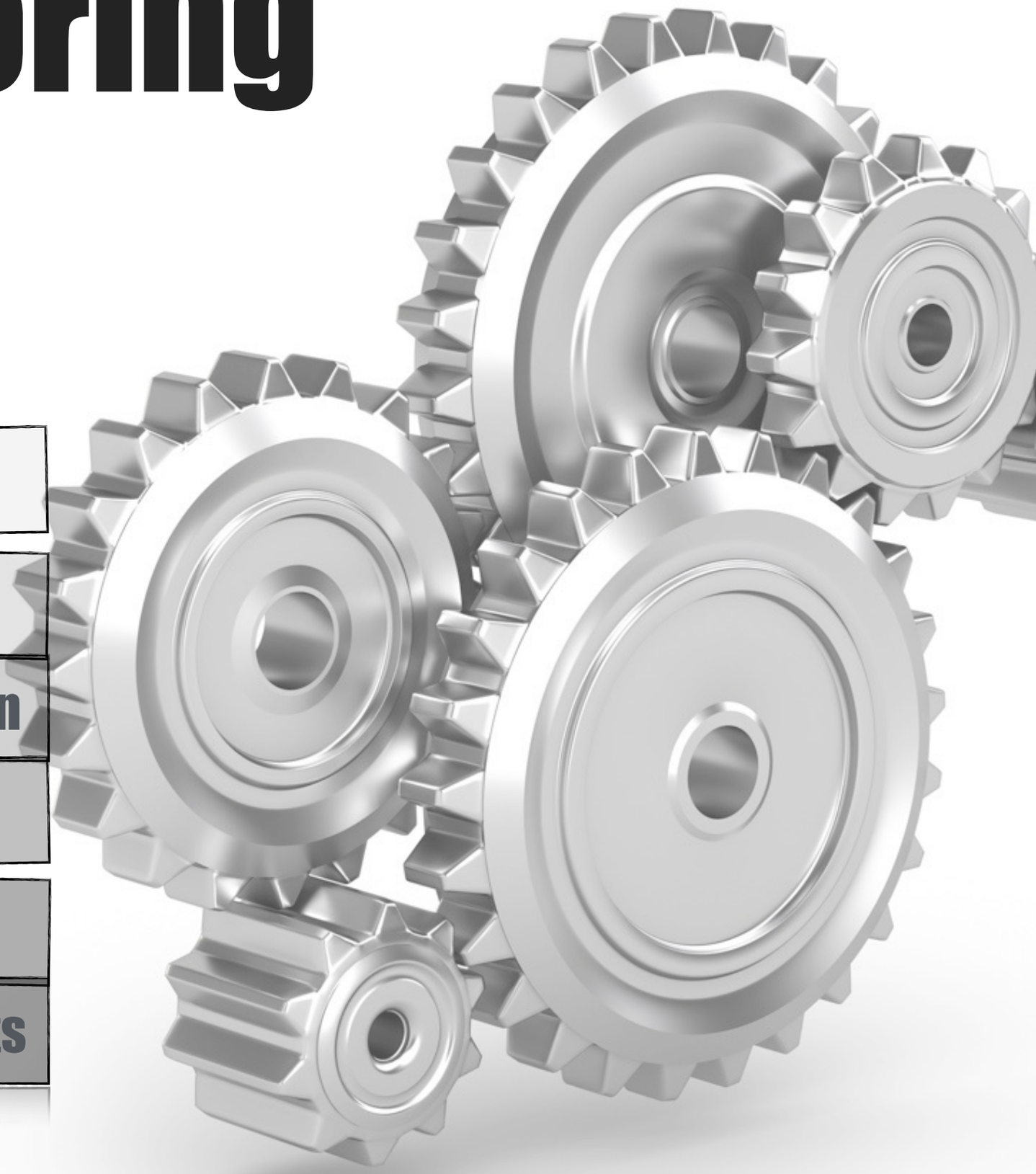


- Where to refactor
- How to refactor?
- Guarantee behaviour preservation
- Apply the refactoring
- Assess its effects on quality
- Consistently modify other artifacts

[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]

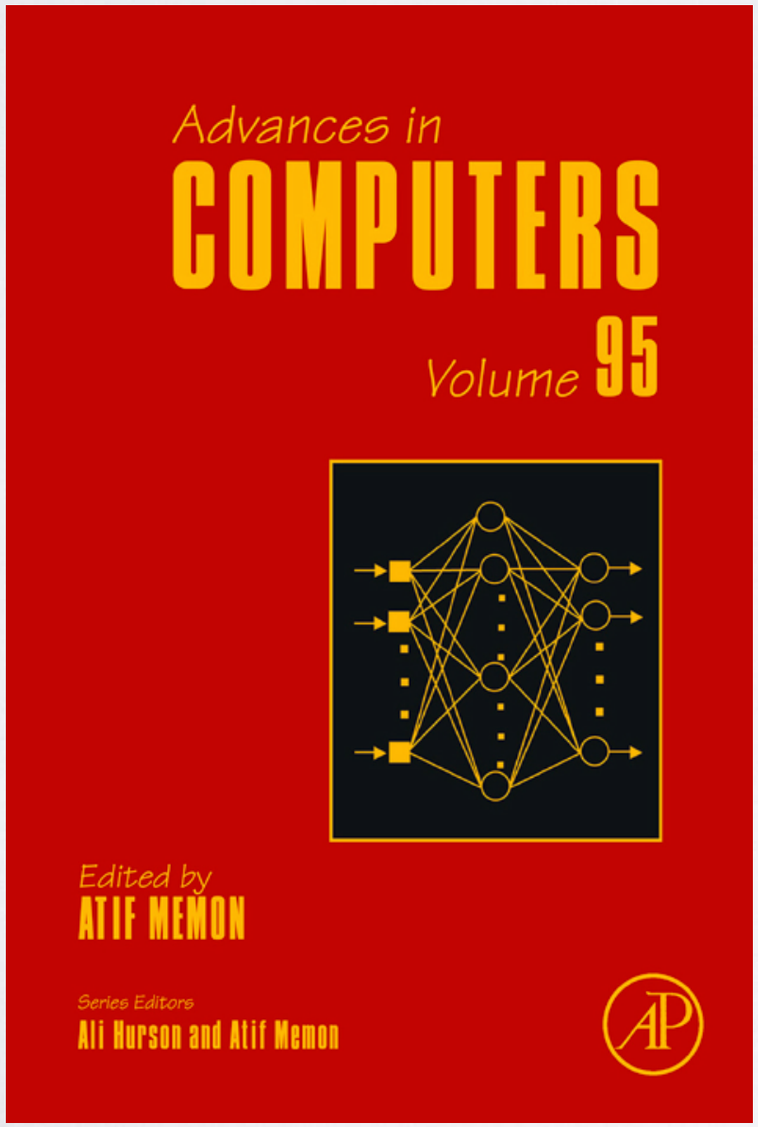
Where to refactor

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‡
*Department of Management and Information Technology, University of Salerno, Fisciano, Italy
†Department of Engineering, University of Sannio, Benevento, Italy
‡Department of Bioscience and Territory, University of Molise, Pesche, Italy

Contents

Advances in Computers, Volume 95
ISSN 0065-2458
http://dx.doi.org/10.1016/B978-0-12-800160-8.00004-8

© 2014 Elsevier Inc.
All rights reserved.

Most approaches use structural information

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 constitute 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: nikos@java.uom.gr, achat@uom.gr.*

Manuscript received 15 Apr. 2008; revised 5 Dec. 2008; accepted 15 Dec. 2008; published online 6 Jan. 2009.
Recommended for acceptance by H. Ossher.
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.

0098-5589/09/$25.00 © 2009 IEEE

Published by the IEEE Computer Society

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

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

0098-5589/10/$26.00 © 2010 IEEE

Published by the IEEE Computer Society

2. Correction is future work.

DECOR

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

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

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

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

1 Introduction

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

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

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

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

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

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

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

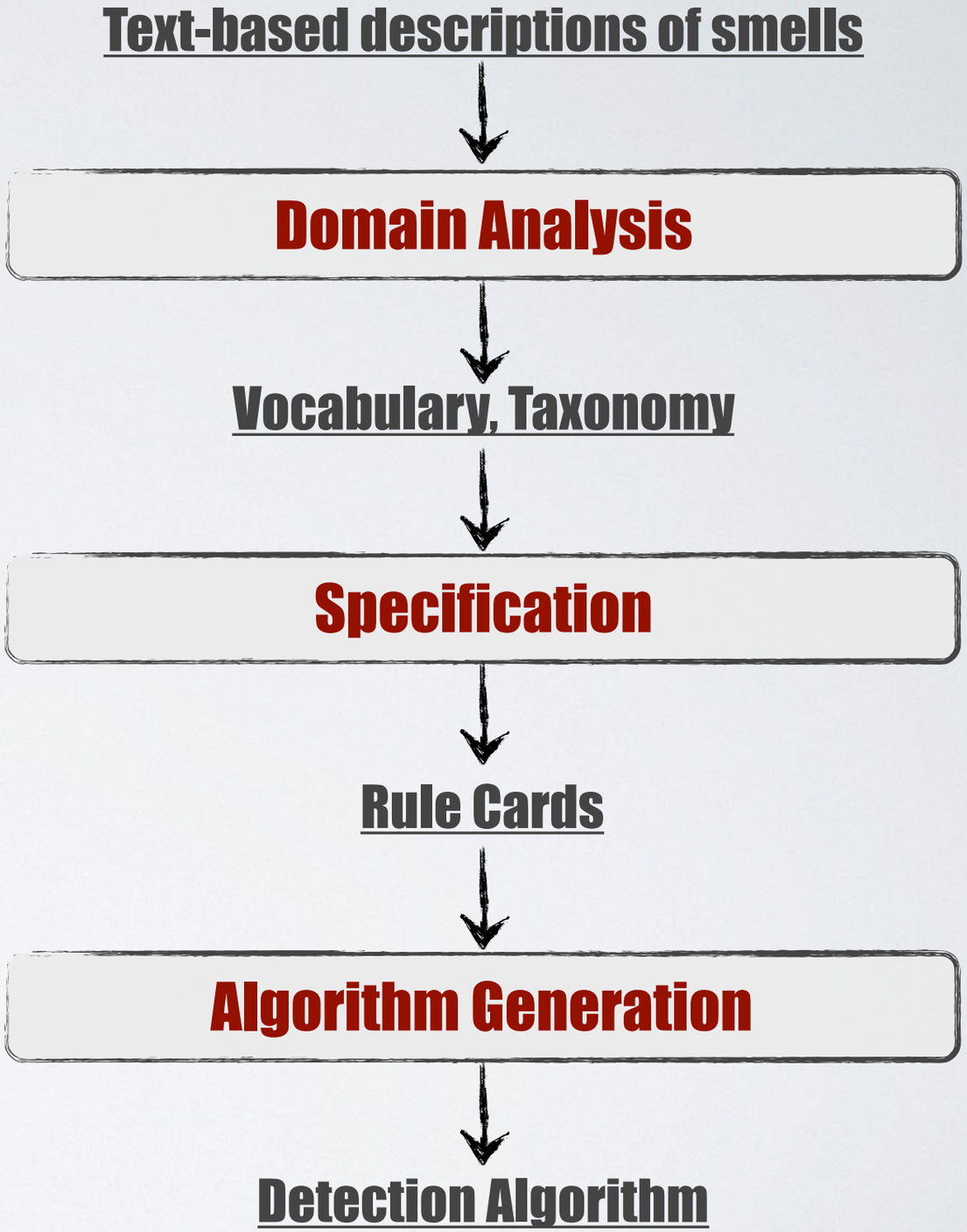
1. This smell, like those presented later on, is really in between implementation and design.

2. Correction is future work.

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

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

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



[Moha et al. TSE 2010]

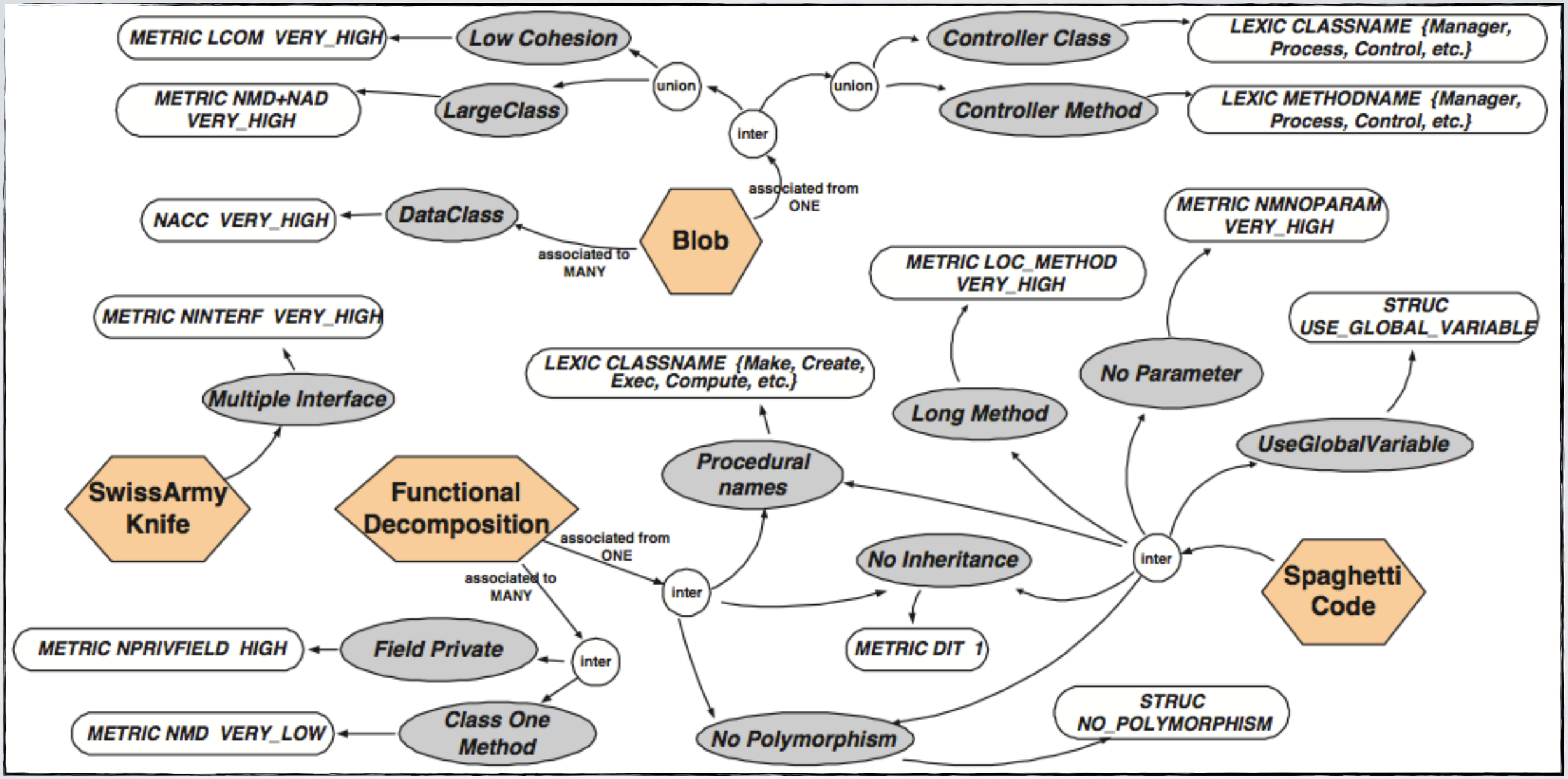
DECOR

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

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

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

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

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

1 INTRODUCTION

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

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

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

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

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

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

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

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

2. Correction is future work.

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

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

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

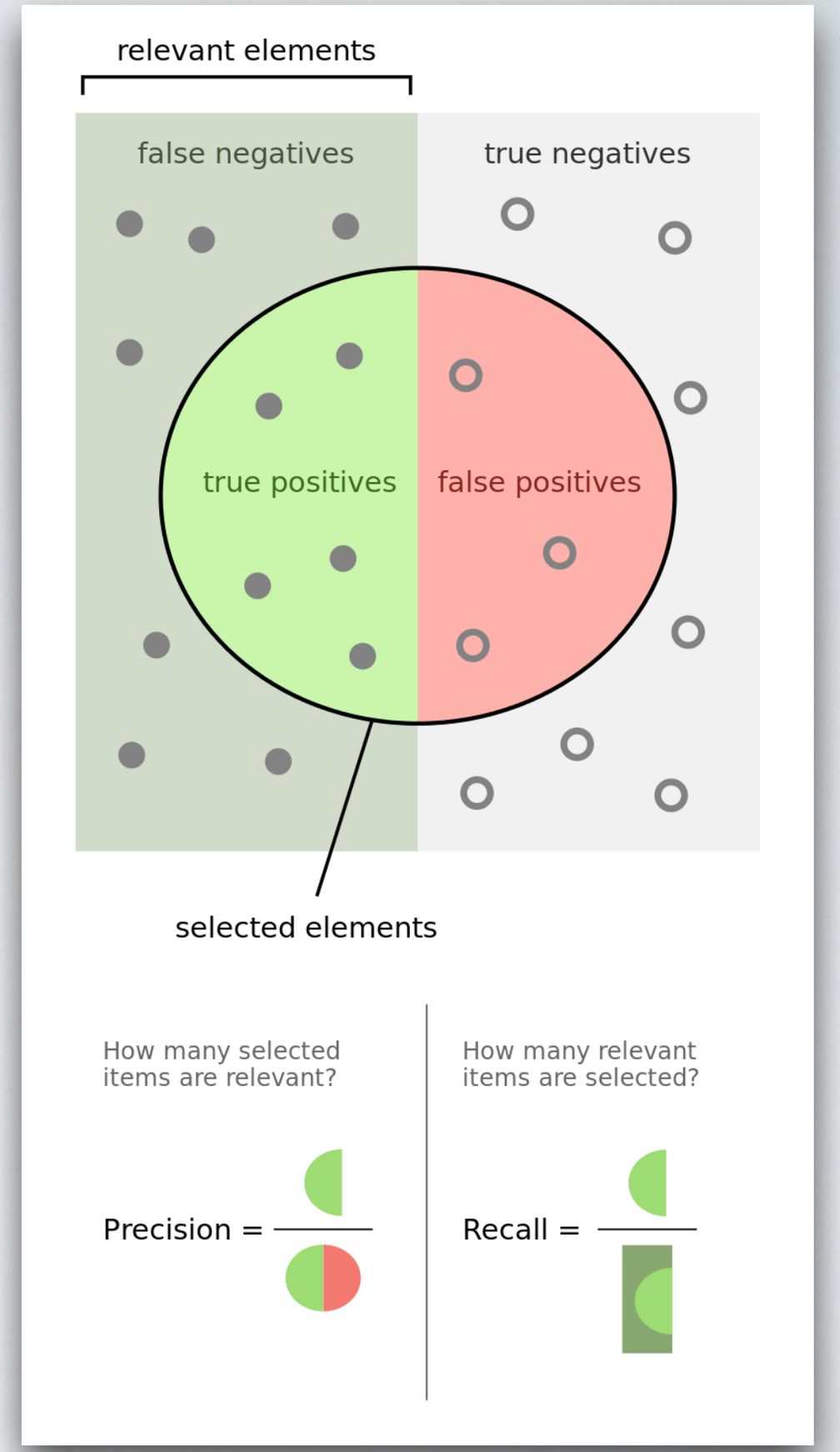
Performances

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

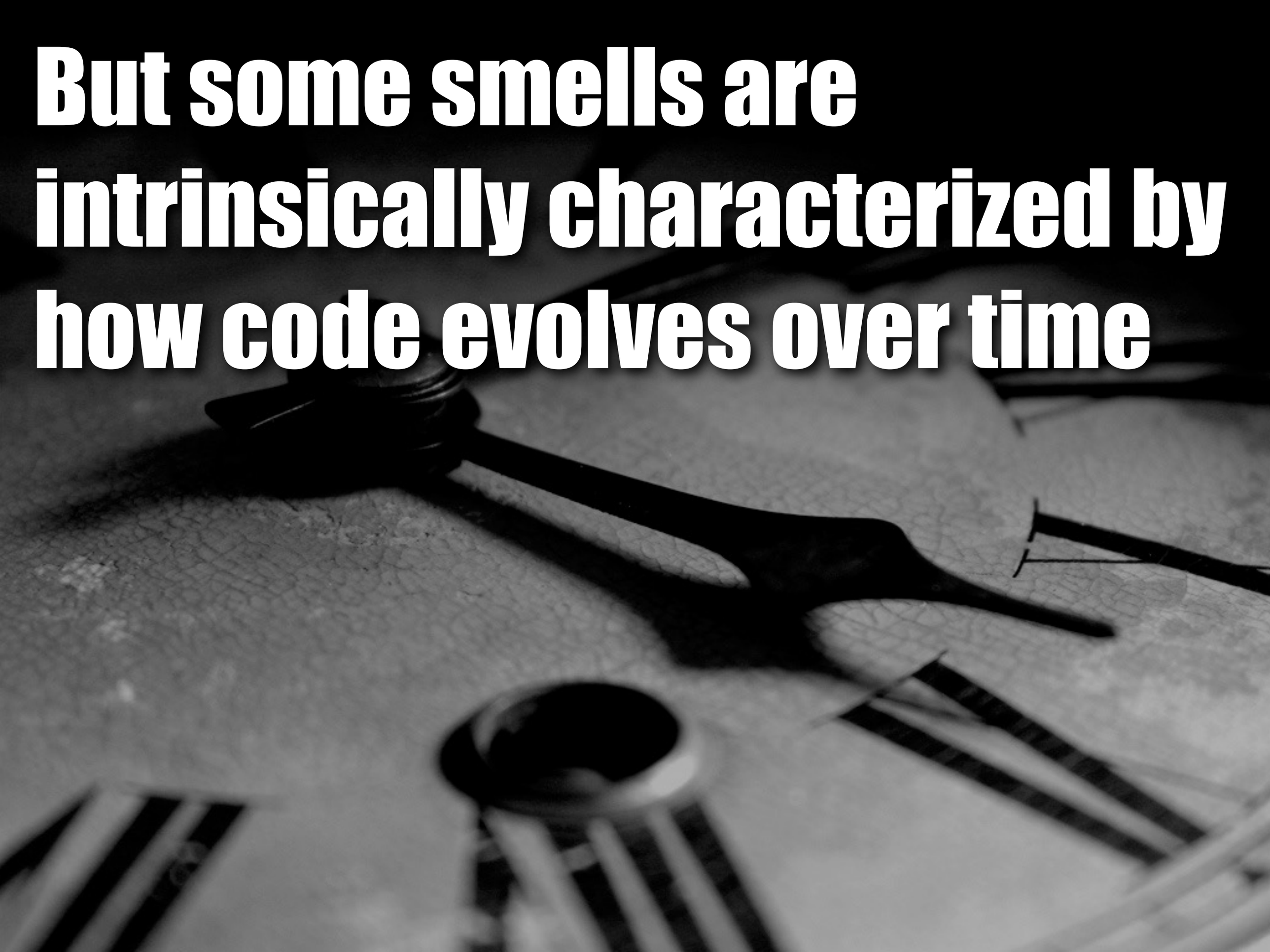
Average Recall: 100%
Average Precision: 60.5%

[Moha et al. TSE 2010]

DECOR


relevant elements
false negatives
true negatives
true positives
false positives
selected elements
How many selected items are relevant?
Precision =
How many relevant items are selected?
Recall =

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

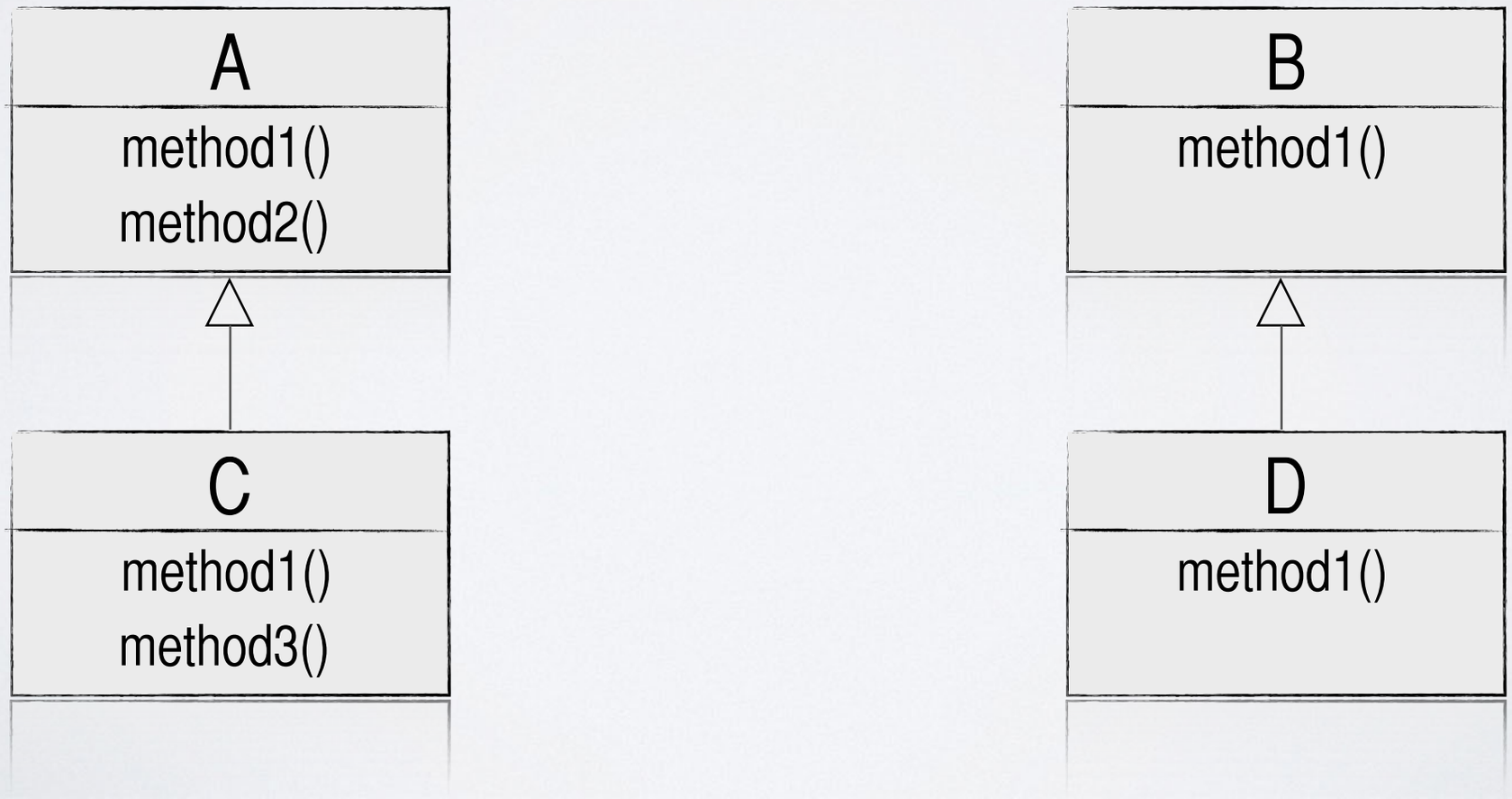


Parallel Inheritance

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

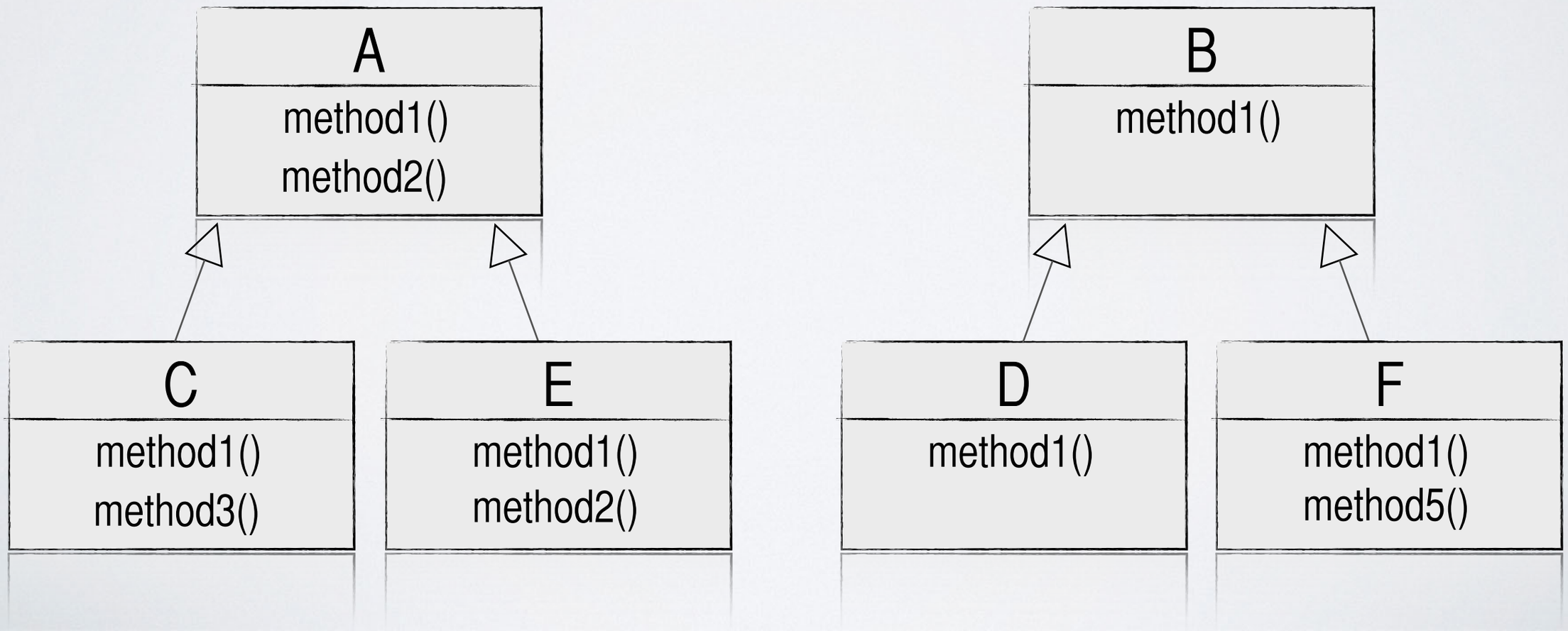
Parallel Inheritance

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.

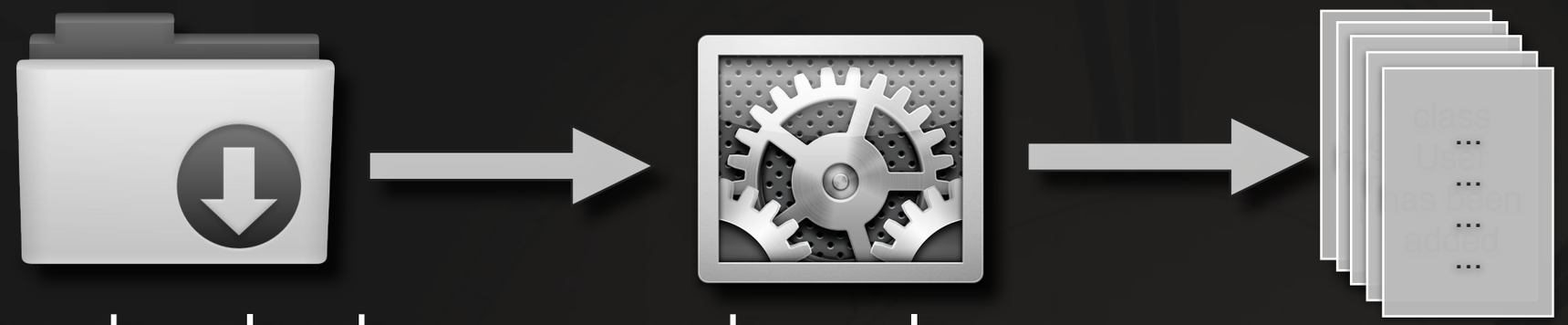
Based on such considerations, we propose an approach, named *H*istorical *I*nformation for *S*mell de*T*ection (HIST), to detect smells based on change history information mined from versioning systems, and, specifically, by analyzing co-changes occurring between source code artifacts. HIST is

- *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

0098-5589 © 2014 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

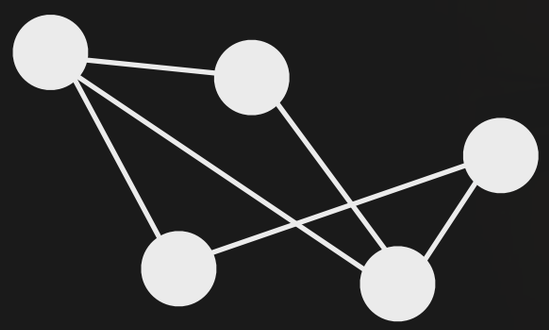
HIST Change History Extractor



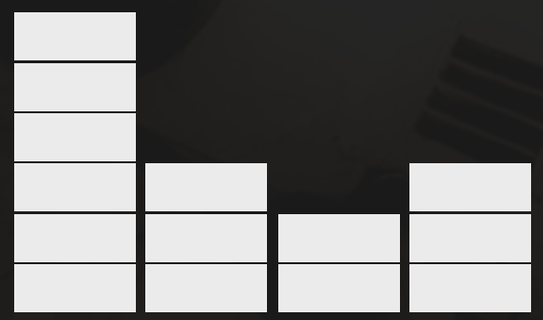
log download

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 *H*istorical *I*nformation for *S*mell de*T*ection (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 n 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 *H*istorical *I*nformation for *S*mell de*T*ection (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

0098-5589 © 2014 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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

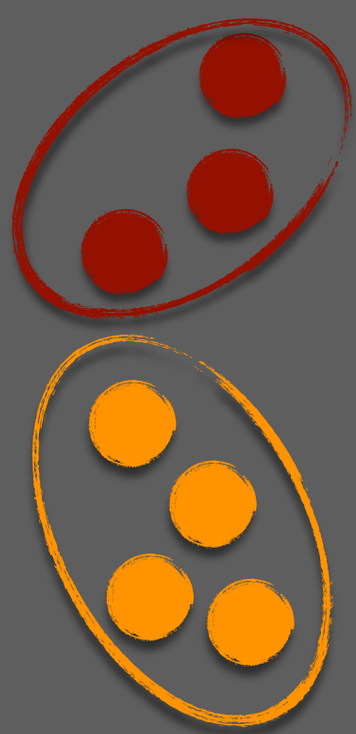
**Solution:
Extract Class Refactoring**

Detection

Classes containing at least two sets of methods such that:

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

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



Code Smells Detector

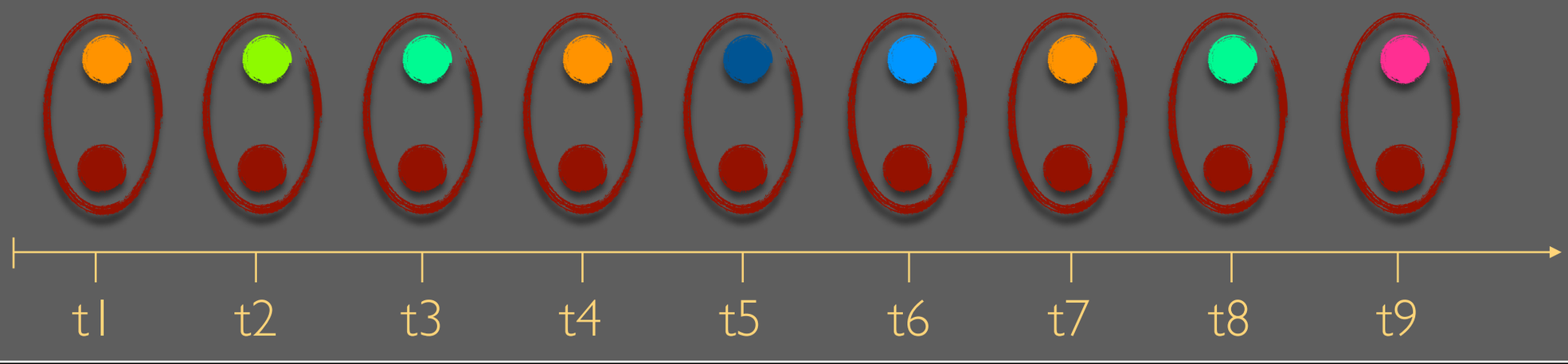
blob

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

Solution:
Extract Class refactoring

Detection

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



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 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

0098-5589 © 2014 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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

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

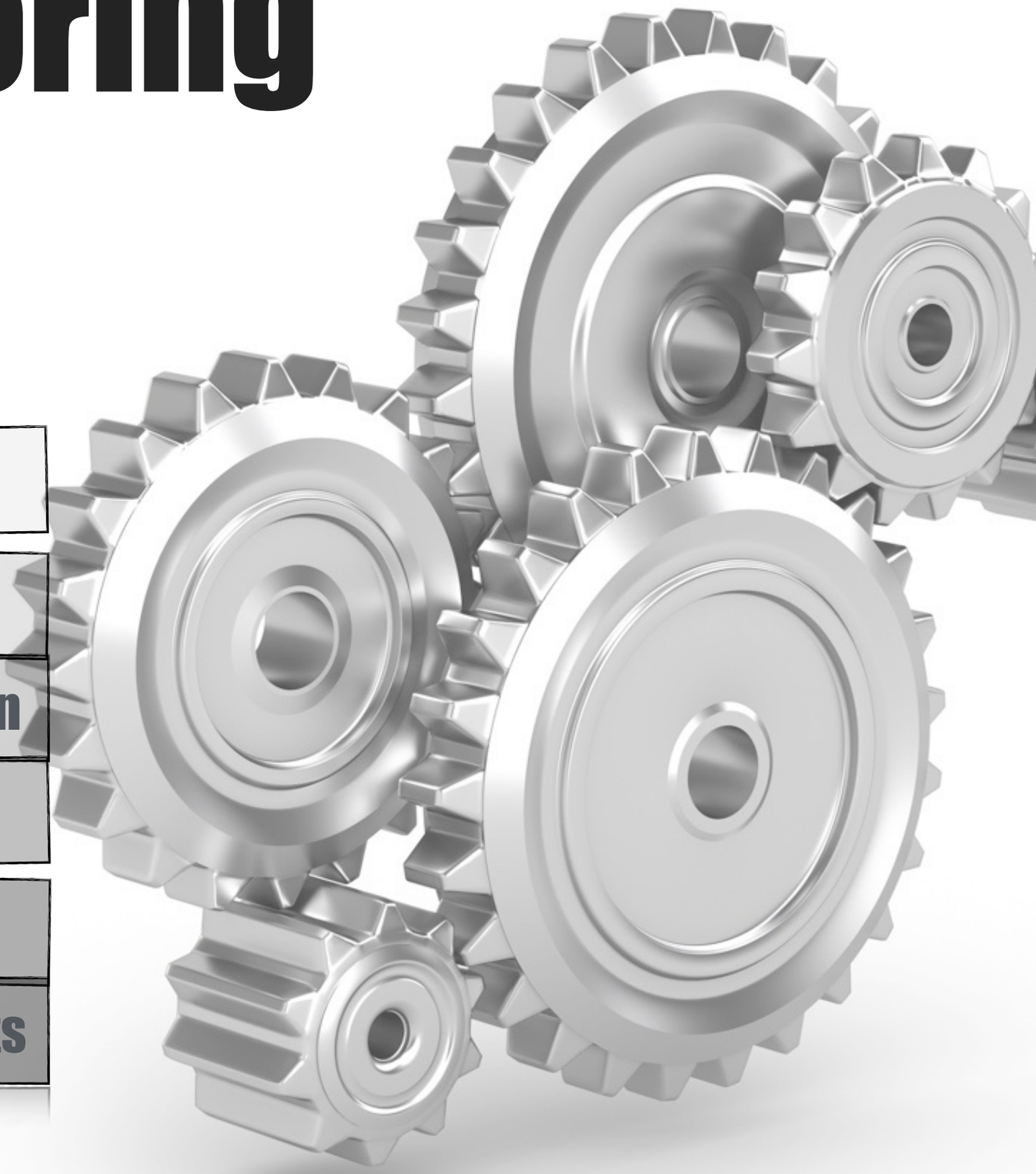
0098-5589 © 2014 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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

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?

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

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

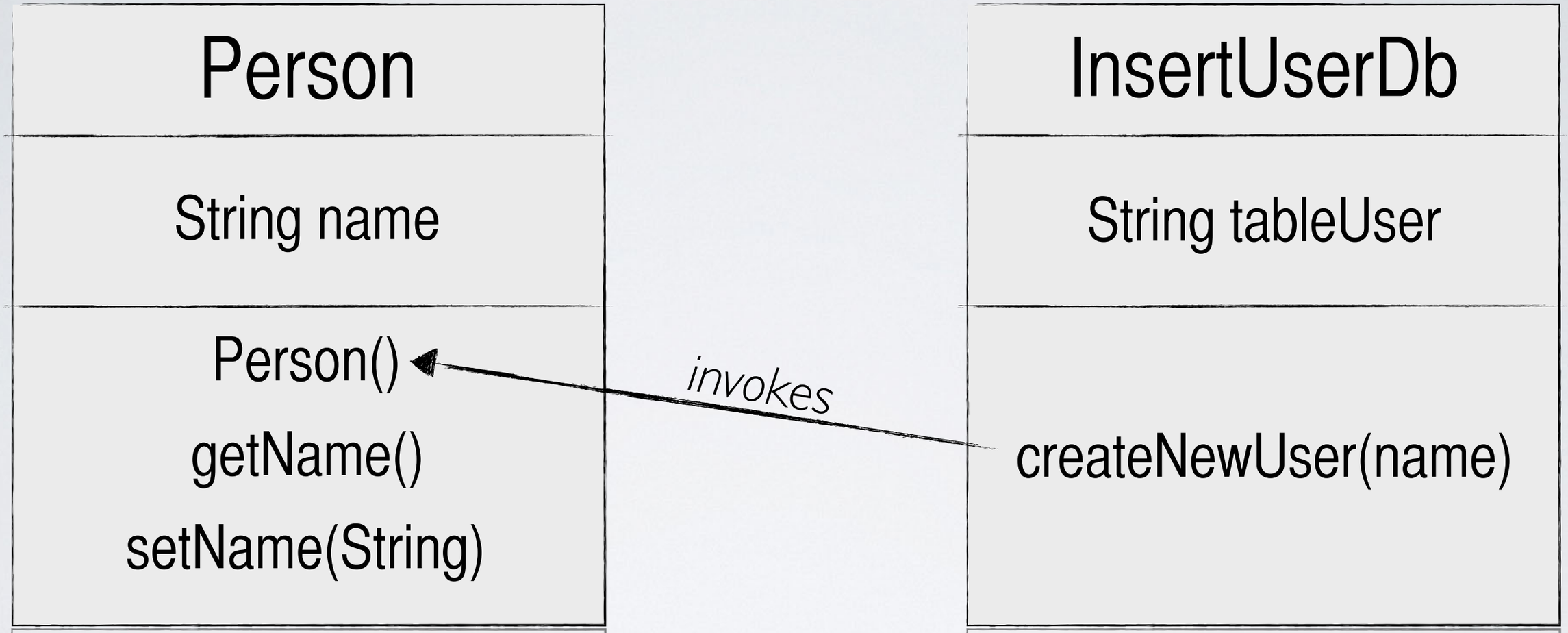
Possible Sources of Information

- Structural
- Dynamic
- Semantic
- Historical

Structural Information

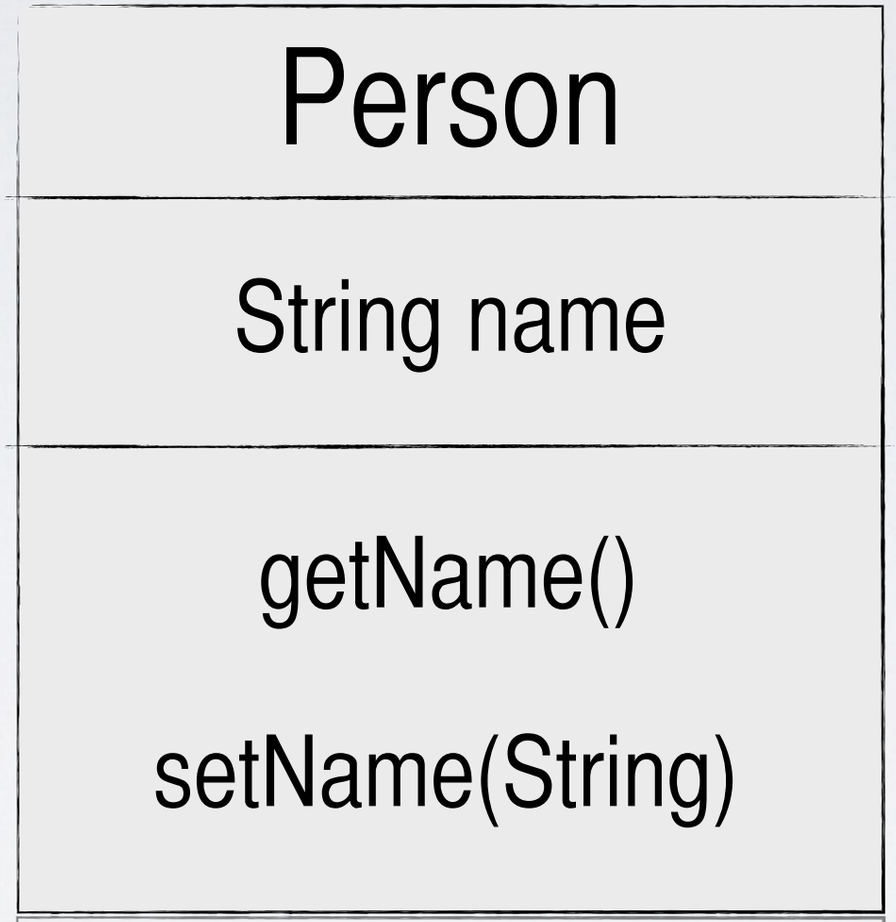
- Calls between Methods
- Shared Attributes
- Inheritance Relationships
- Original Design

Calls between methods



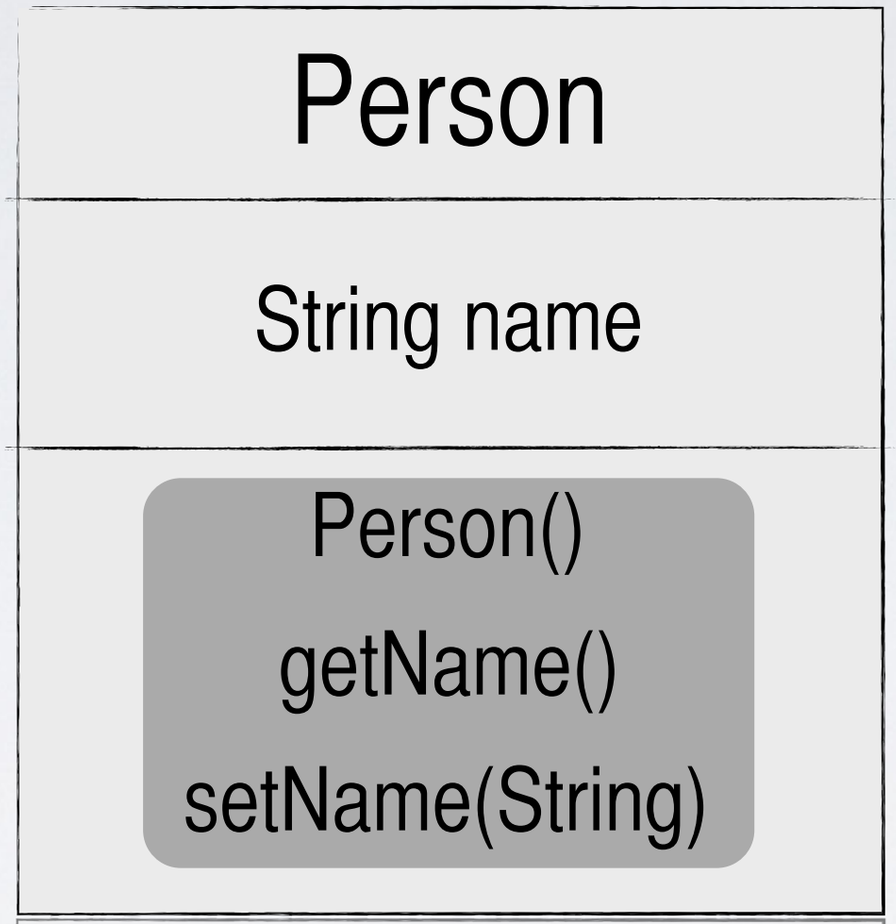
Methods (Classes) having calls between them are more likely to implement similar resposibilities

Shared Attributes



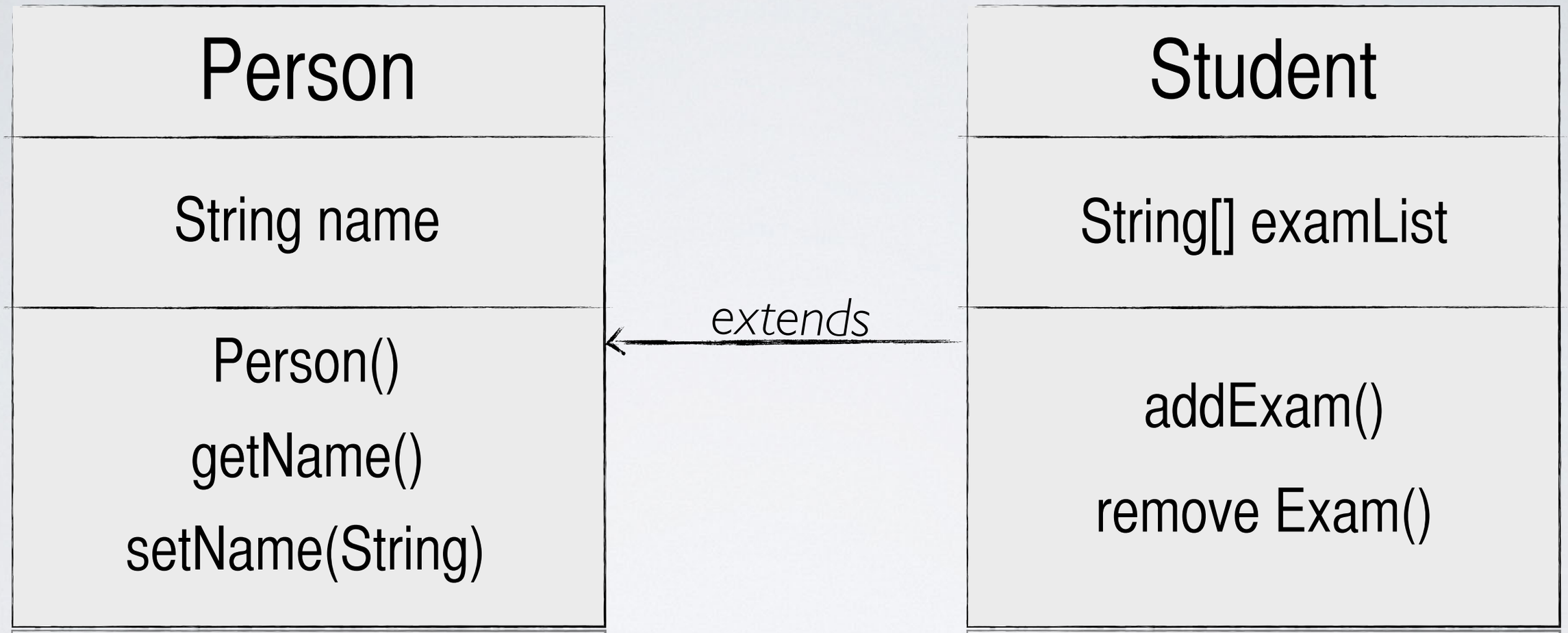
Methods sharing attributes are more likely to implement similar resposibilities

Original Design



Methods (Classes) grouped together by original developers are more likely to be related one each other

Inheritance Relationships



Classes having inheritance relationships are more likely to implement similar resposibilities

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

Semantic 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);
}
```

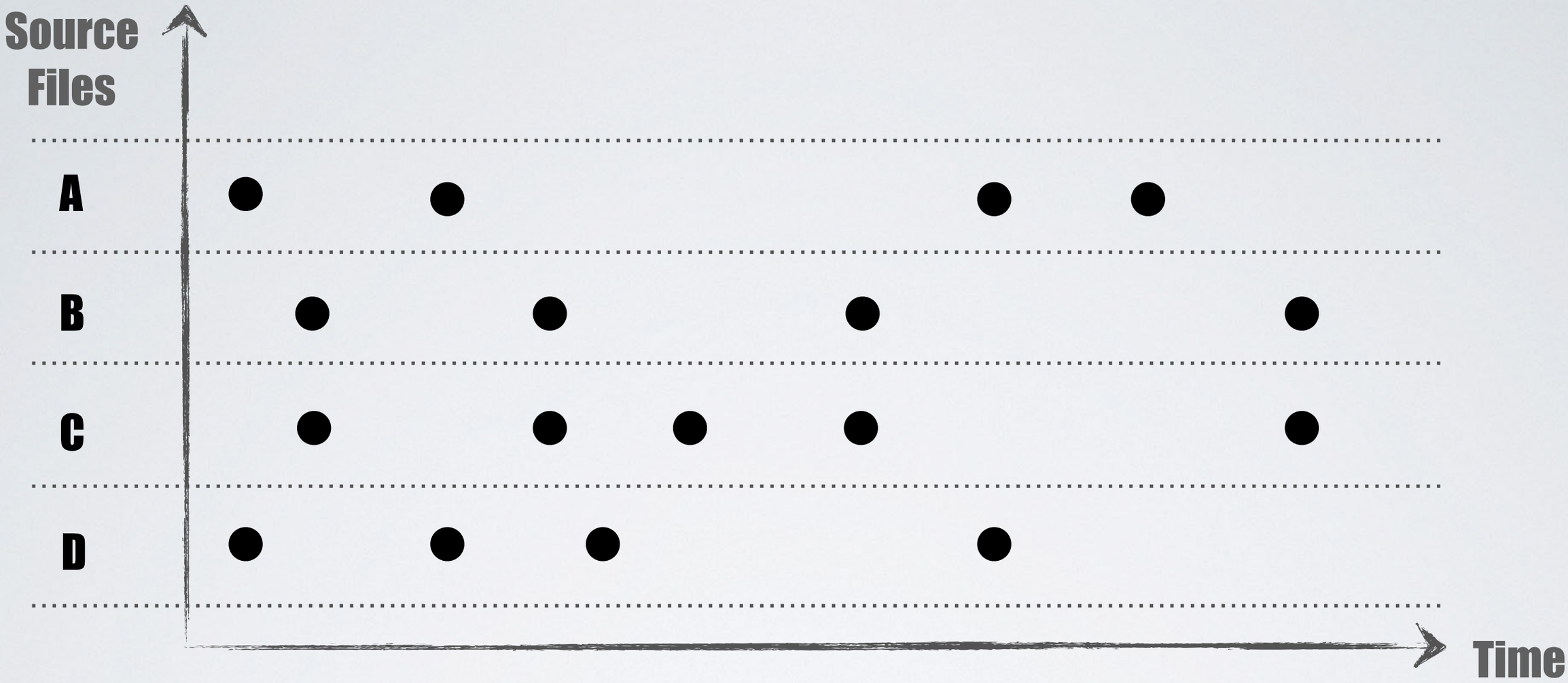
```
/* Delete an user from the system.
 *  @param pUser: the user to delete.*/
public void delete(User pUser) {

   connect = DBConnection.getConnection();

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

   executeOperation(connect, sql);
 }
```

Semantic 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);
}
```

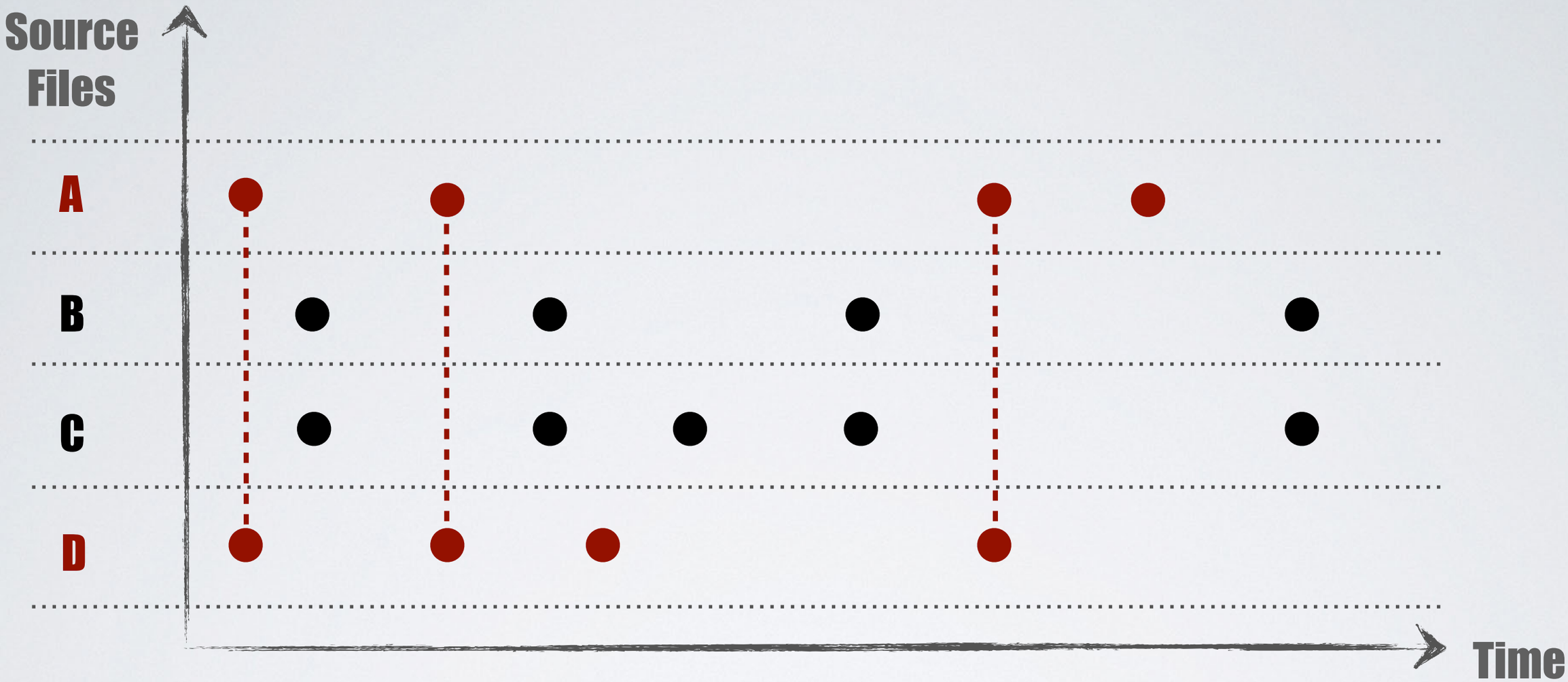
```
/* Delete an user from the system.
 *  @param pUser: the user to delete.*/
public void delete(User pUser) {

   connect = DBConnection.getConnection();

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

   executeOperation(connect, sql);
 }
```

Historical Information



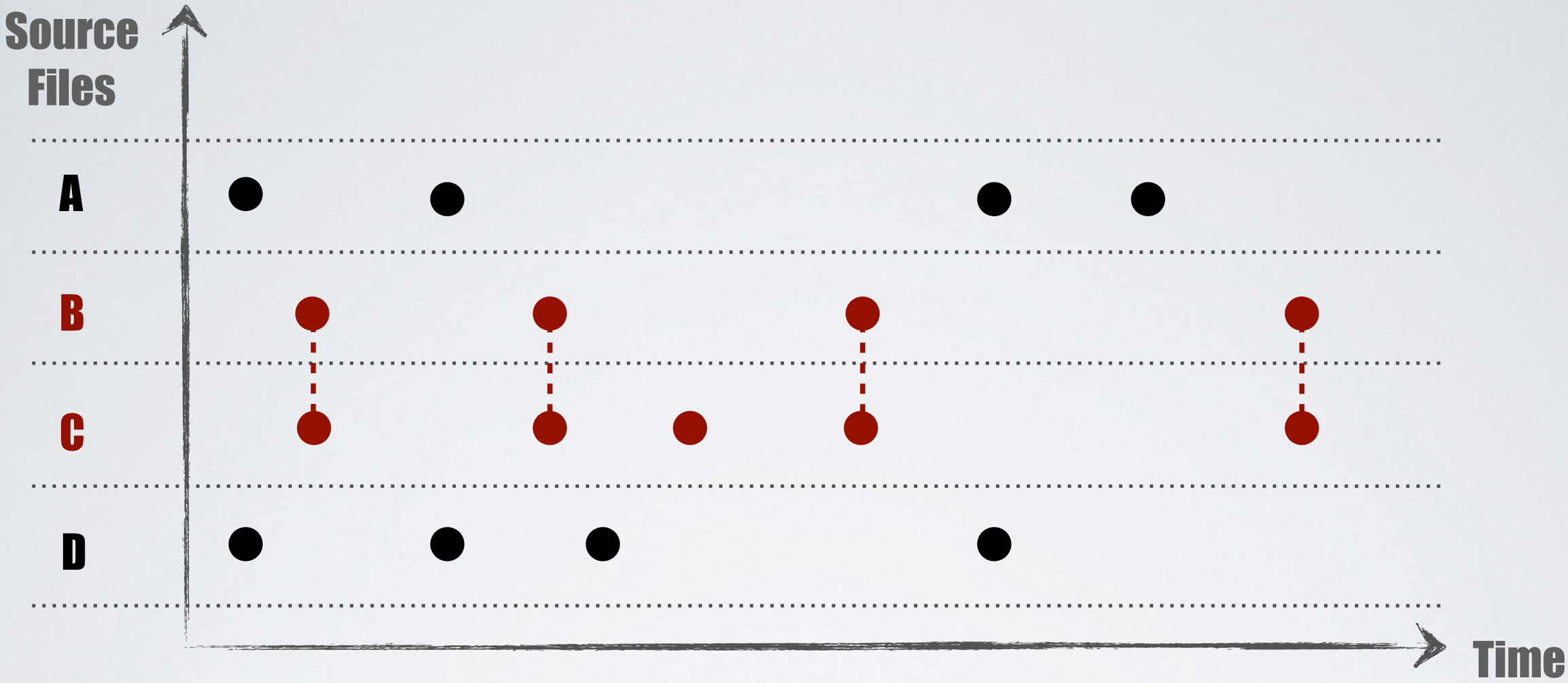
Code components often changing together during software history are more likely to implement similar resposibilities

Historical Information



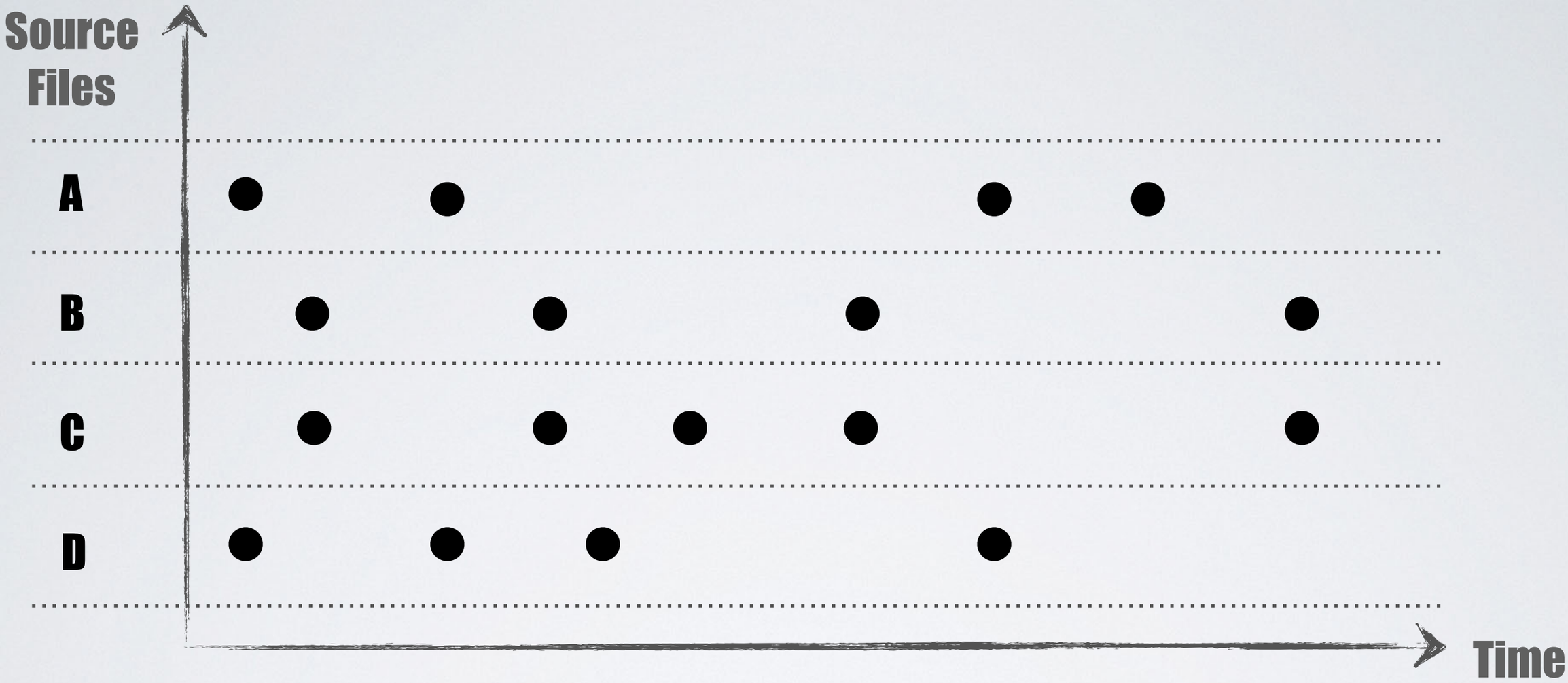
Code components often changing together during software history are more likely to implement similar resposibilities

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

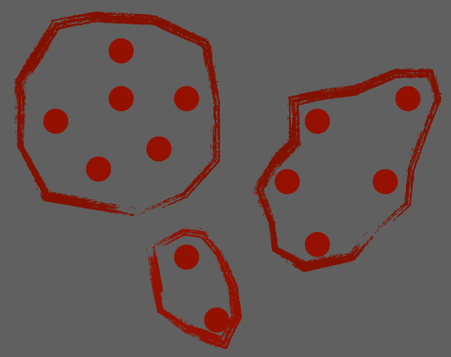
During software evolution change is the rule rather than the exception [27]. Continuous modifications in the environment and requirements drive software evolution. Unfortunately, programmers do not always have the necessary time to make sure that the changes conform to good design practices. In consequence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

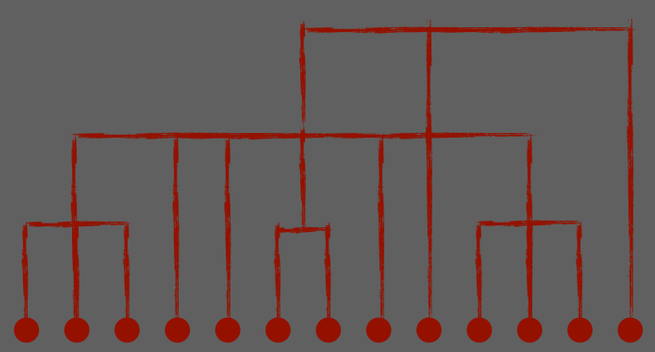
Andrian Marcus
Wayne State University, Detroit MI (USA), e-mail: amarcus@wayne.edu

Rocco Oliveto
University of Molise, Pesche (Italy), e-mail: rocco.oliveto@unimol.it



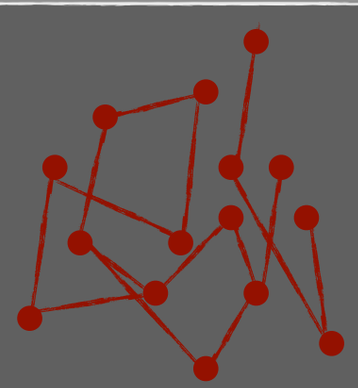
Partitioning algorithms

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.



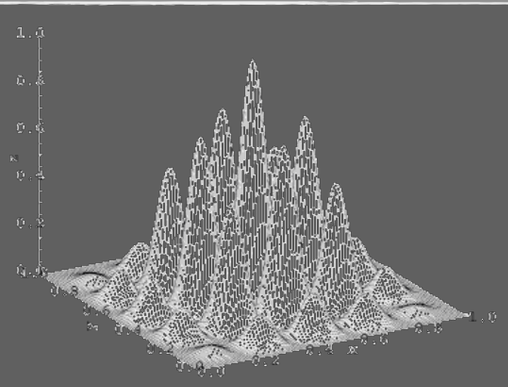
Hierarchical algorithms

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.

```
if a > b then
```

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 84 (2011) 397–414

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures

Gabriele Bavota*, Andrea De Lucia, Rocco Oliveto

Department of Mathematics and Informatics, University of Salerno, via Ponte don Melillo, 84084 Fisciano, SA, Italy

ARTICLE INFO

Article history:
Received 24 May 2010
Received in revised form
23 November 2010
Accepted 25 November 2010
Available online 4 December 2010

Keywords:
Extract Class refactoring
Cohesion
Coupling
Graph theory
MaxFlow-MinCut
Empirical studies

ABSTRACT

Approaches for improving class cohesion identify refactoring opportunities using metrics that capture structural relationships between the methods of a class, e.g., attribute references. Semantic metrics, e.g., C3 metric, have also been proposed to measure class cohesion, as they seem to complement structural metrics. However, until now semantic relationships between methods have not been used to identify refactoring opportunities. In this paper we propose an Extract Class refactoring method based on graph theory that exploits structural and semantic relationships between methods. The empirical evaluation of the proposed approach highlighted the benefits provided by the combination of semantic and structural measures and the potential usefulness of the proposed method as a feature for software development environments.

© 2010 Elsevier Inc. All rights reserved.

1. Introduction

Object-Oriented Programming (OOP) is also responsibility-based programming, where the programmer defines the different responsibilities of the code, and assigns these responsibilities to classes and methods (Coad and Yourdon, 1991; Fowler, 1999). Unfortunately, during software development the classes of a system undergo continuous modifications making the source code more complex and drifting away from its original design. In particular, due to strict deadlines programmers do not always have a bunch of time to make sure everything conforms to OOP guidelines. In many cases, programmers add a responsibility to a class feeling that it is not required to include it in a separate class. However, when the added responsibility grows and breeds, the class becomes too complex and its quality deteriorates (Fowler, 1999). It is worth noting that a class with more than one responsibility (it performs operations that should be done by two or more classes) is difficult to comprehend, thus increasing (i) the overall maintenance effort (Basili et al., 1995; Binkley and Schach, 1998; Briand et al., 1999b,a; Chidamber and Kemerer, 1994) and (ii) the risk to introduce a bug (Gyimóthy et al., 2005; Liu et al., 2009; Marcus et al., 2008).

The degree to which the responsibilities of a class (or module) form a meaningful unit are measured by class cohesion (Coad and Yourdon, 1991). Classes with unrelated methods (different responsibilities) have low cohesion and they often need to be refactored (Fowler, 1999; Mens and Tourwe, 2004) by distributing some of their responsibilities to new classes, thus reducing their complexity and improving their cohesion. In particular, *Extract Class* refactoring (Fowler, 1999) is a common technique to split a class with many responsibilities into different classes.

Several semi-automatic approaches have been proposed to improve the cohesion of a class by identifying refactoring opportunities (Atkinson and King, 2005; O'Keeffe and O'Cinneide, 2006; Maruyama and Shima, 1999; Moore, 1996). In general, such approaches use structural measures that capture relationships between methods, such as attribute references and method calls, to guide refactoring operations on the source code. Indeed, most of the metrics proposed to measure class cohesion and to guide refactoring are based on structural information of the source code (see e.g., O'Keeffe and O'Cinneide, 2006; Maruyama and Shima, 1999). Recently, semantic metrics have also been proposed to measure class cohesion (Marcus et al., 2008). These metrics use semantic information retrieved from comments and identifiers and seem to complement structural metrics (Marcus et al., 2008). However, until now they have not been used to guide refactoring activities.

The orthogonality between structural and semantic cohesion metrics motivates our work. We conjecture that better refactoring opportunities might be identified using a combination of structural and semantic cohesion measures. In particular, we propose a novel approach to support Extract Class refactoring that identifies disjoined sets of strongly related methods in a given class analysing the

* Corresponding author. Tel.: +39 089963381; fax: +39 089963303.
E-mail addresses: gbavota@unisa.it (G. Bavota), adelucia@unisa.it (A. De Lucia), roliveto@unisa.it (R. Oliveto).

0164-1212/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2010.11.918

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

Empir Software Eng (2014) 19:1617–1664
DOI 10.1007/s10664-013-9256-x

Automating extract class refactoring: an improved method and its evaluation

Gabriele Bavota · Andrea De Lucia · Andrian Marcus · Rocco Oliveto

Published online: 4 May 2013
© Springer Science+Business Media New York 2013

Abstract During software evolution the internal structure of the system undergoes continuous modifications. These continuous changes push away the source code from its original design, often reducing its quality, including class cohesion. In this paper we propose a method for automating the *Extract Class* refactoring. The proposed approach analyzes (structural and semantic) relationships between the methods in a class to identify chains of strongly related methods. The identified method chains are used to define new classes with higher cohesion than the original class, while preserving the overall coupling between the new classes and the classes interacting with the original class. The proposed approach has been first assessed in an artificial scenario in order to calibrate the parameters of the approach. The data was also used

Communicated by: Arie van Deursen

G. Bavota · A. De Lucia
Software Engineering Lab, University of Salerno, Via ponte don Melillo, 84084, Fisciano SA, Italy

G. Bavota
e-mail: gbavota@unisa.it
URL: http://www.dmi.unisa.it/people/bavota/www

A. De Lucia
e-mail: adelucia@unisa.it
URL: http://www.dmi.unisa.it/people/delucia/www

A. Marcus
SEVERE Group, Department of Computer Science, Wayne State University, 5057 Woodward Ave, Suite 14101.1, Detroit, MI 48202, USA
e-mail: amarcus@wayne.edu
URL: http://www.cs.wayne.edu/~amarcus/

R. Oliveto (✉)
Department of Bioscience and Territory, University of Molise, C. da Fonte Lappone, 86090, Pesche IS, Italy
e-mail: rocco.oliveto@unimol.it
URL: http://www.distat.unimol.it/people/oliveto/Home.html

Springer

Supported Refactoring Operation: Extract Class Refactoring

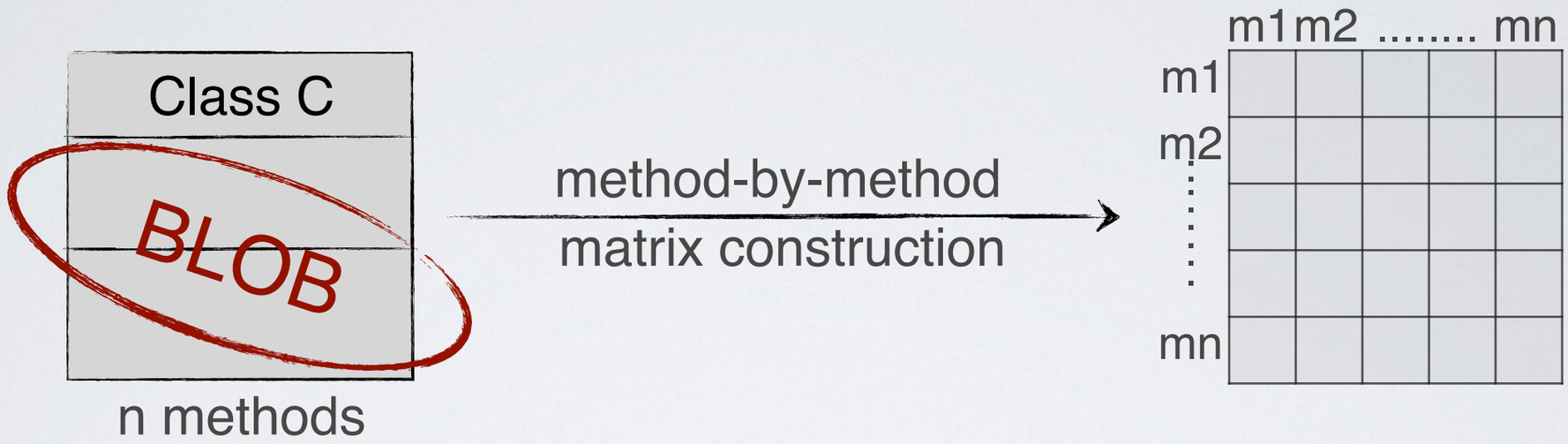
Exploited Information: Shared attributes, method calls, semantic similarity

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

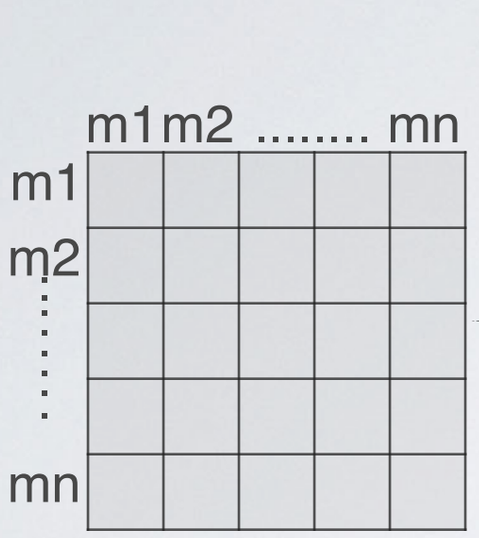
CIM

Call-based Interaction between Methods

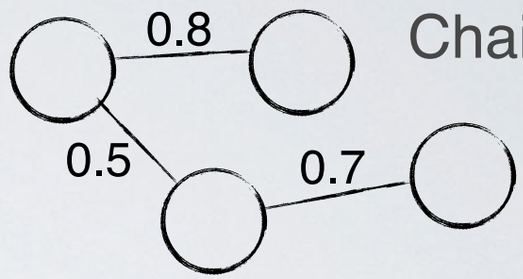
CSM

Conceptual Similarity between Methods

GRAPH theory-based ECR

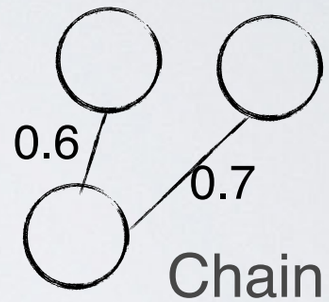


method-by-method matrix filtering

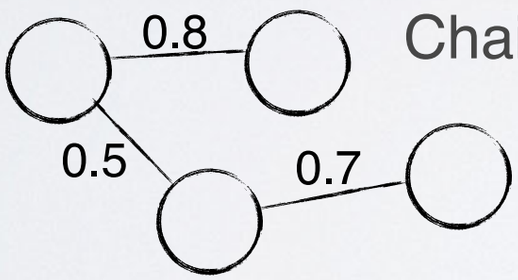


Chain 1

Chain 3

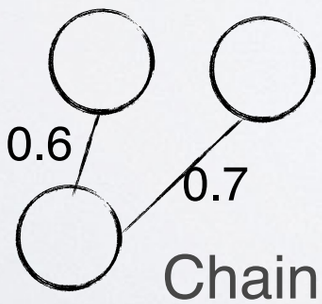


Chain 2



Chain 1

Chain 3



Chain 2

merging trivial chains

Candidate Class 1

Chain 1

Candidate Class 2

Chain 2
Chain 3

The Journal of Systems and Software 85 (2012) 2241–2260

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

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

Identification and application of Extract Class refactorings in object-oriented systems

Marios Fokaefs [a,*], Nikolaos Tsantalis [a], Eleni Stroulia [a], Alexander Chatzigeorgiou [b]

[a] Department of Computing Science, University of Alberta, Edmonton, Canada
[b] Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece

ARTICLE INFO

Article history:
Received 12 December 2010
Received in revised form 17 March 2012
Accepted 8 April 2012
Available online 25 April 2012

MSC:
10
10.040
10.060
10.070

Keywords:
Refactoring
Software reengineering
Object-oriented programming
Clustering

ABSTRACT

Refactoring is recognized as an essential practice in the context of evolutionary and agile software development. Recognizing the importance of the practice, modern IDEs provide some support for low-level refactorings. A notable exception in the list of supported refactorings is the "Extract Class" refactoring, which is conceived to simplify large, complex, unwieldy and less cohesive classes.

In this work, we describe a method and a tool, implemented as an Eclipse plugin, designed to fulfill exactly this need. Our method involves three steps: (a) recognition of Extract Class opportunities, (b) ranking of the identified opportunities in terms of the improvement each one is anticipated to bring about to the system design, and (c) fully automated application of the refactoring chosen by the developer. The first step relies on an agglomerative clustering algorithm, which identifies cohesive sets of class members within the system classes. The second step relies on the Entity Placement metric as a measure of design quality. Through a set of experiments we have shown that the tool is able to identify and extract new classes that developers recognize as "coherent concepts" and improve the design quality of the underlying system.

© 2012 Elsevier Inc. All rights reserved.

1. Introduction

Evolutionary software development is the most broadly adopted lifecycle process today. Software evolves throughout its lifecycle, even past its release, and, as a result, the as-is design of the system usually ends up deviating from its original rationale and violating design principles. Such violations manifest themselves as "bad smells" (Fowler et al., 1999) and refactoring becomes necessary to eliminate them (Opdyke, 1992). Refactoring is the process of introducing behavior preserving restructurings to the code, in order to improve its design and enable it to support further development.

This work is motivated by a specific bad smell called "God Class" (Fowler et al., 1999). In principle, a class should implement only one concept (Martin, 2003) and should only change when the concept it encapsulates evolves. The violation of this principle results in large, complex, unwieldy, inelegant, less cohesive and difficult to understand and maintain "God Classes". Generally, there are two types of such classes: some hold a lot of the system's data in terms of number of attributes ("Data God Classes" or "Lazy Classes") and others implement a great portion of the system's functionality in terms of many and frequently complex methods ("Behavioral God Classes"). In the first case, developers can redistribute the attributes of the "God Class" or move functionality (i.e. methods) from other classes closer to the data. In the second case, they can either move functionality from the "God Class" closer to the data of other classes or simplify the class by extracting a cohesive and independent piece of functionality (Fowler et al., 1999; Demeyer et al., 2002). The latter is a refactoring called "Extract Class".

Our work is not trying to identify "God Classes", but rather Extract Class refactoring opportunities in order to decompose large classes. The proposed method recognizes coherent packages of data and behaviors which if extracted into a new class would result in improving the overall system design and, at the request of the developer, automatically applies the "Extract Class" refactoring. To identify Extract Class opportunities, our method employs a clustering algorithm, as clustering has long been used for software remodularization (Tzerpos and Holt, 1998; Wiggerts, 1997). More specifically, the intuition behind using clustering in this problem is that clusters may represent cohesive groups of class members (methods and attributes) that have a distinct functionality and

* Corresponding author. Tel.: +1 780 886 6893.
E-mail addresses: fokaefs@ualberta.ca (M. Fokaefs), tsantalis@ualberta.ca (N. Tsantalis), stroulia@ualberta.ca (E. Stroulia), achat@uom.gr (A. Chatzigeorgiou).

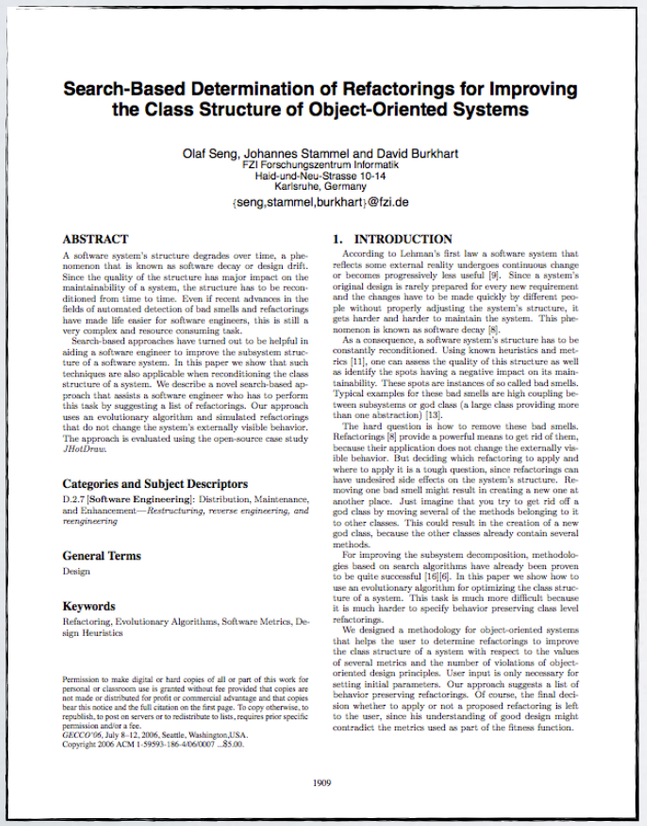
0164-1212/$ - see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2012.04.013

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



Search-Based Determination of Refactorings for Improving the Class Structure of Object-Oriented Systems

Olaf Seng, Johannes Stammel and David Burkhart
FZI Forschungszentrum Informatik
Haid-und-Neu-Strasse 10-14
Karlsruhe, Germany
{seng,stammel,burkhart}@fzi.de

ABSTRACT

A software system's structure degrades over time, a phenomenon that is known as software decay or design drift. Since the quality of the structure has major impact on the maintainability of a system, the structure has to be reconditioned from time to time. Even if recent advances in the fields of automated detection of bad smells and refactorings have made life easier for software engineers, this is still a very complex and resource consuming task.

Search-based approaches have turned out to be helpful in aiding a software engineer to improve the subsystem structure of a software system. In this paper we show that such techniques are also applicable when reconditioning the class structure of a system. We describe a novel search-based approach that assists a software engineer who has to perform this task by suggesting a list of refactorings. Our approach uses an evolutionary algorithm and simulated refactorings that do not change the system's externally visible behavior. The approach is evaluated using the open-source case study *JHotDraw*.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Design

Keywords

Refactoring, Evolutionary Algorithms, Software Metrics, Design Heuristics

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'06, July 8–12, 2006, Seattle, Washington,USA.
Copyright 2006 ACM 1-59593-186-4/06/0007 ...$5.00.

1. INTRODUCTION

According to Lehman's first law a software system that reflects some external reality undergoes continuous change or becomes progressively less useful [9]. Since a system's original design is rarely prepared for every new requirement and the changes have to be made quickly by different people without properly adjusting the system's structure, it gets harder and harder to maintain the system. This phenomenon is known as software decay [8].

As a consequence, a software system's structure has to be constantly reconditioned. Using known heuristics and metrics [11], one can assess the quality of this structure as well as identify the spots having a negative impact on its maintainability. These spots are instances of so called bad smells. Typical examples for these bad smells are high coupling between subsystems or god class (a large class providing more than one abstraction) [13].

The hard question is how to remove these bad smells. Refactorings [8] provide a powerful means to get rid of them, because their application does not change the externally visible behavior. But deciding which refactoring to apply and where to apply it is a tough question, since refactorings can have undesired side effects on the system's structure. Removing one bad smell might result in creating a new one at another place. Just imagine that you try to get rid off a god class by moving several of the methods belonging to it to other classes. This could result in the creation of a new god class, because the other classes already contain several methods.

For improving the subsystem decomposition, methodologies based on search algorithms have already been proven to be quite successful [16][6]. In this paper we show how to use an evolutionary algorithm for optimizing the class structure of a system. This task is much more difficult because it is much harder to specify behavior preserving class level refactorings.

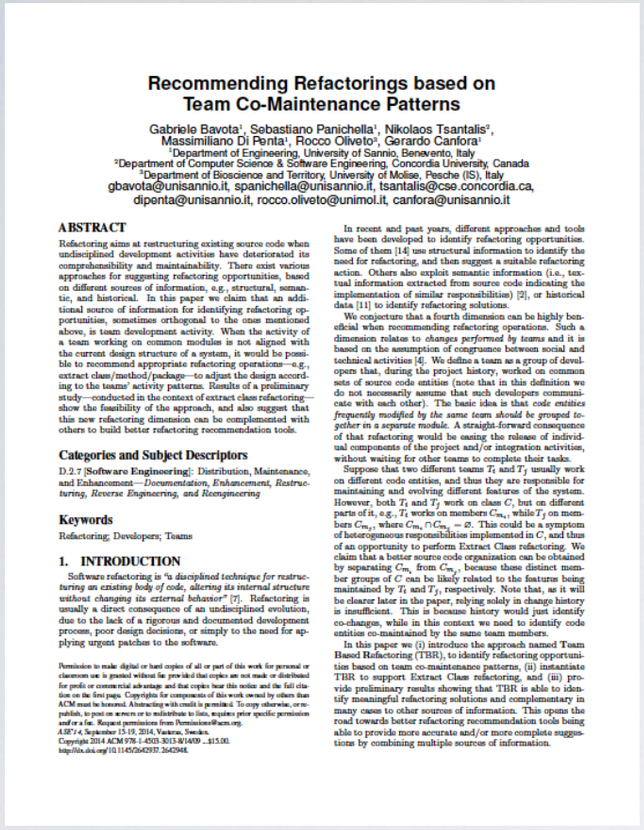
We designed a methodology for object-oriented systems that helps the user to determine refactorings to improve the class structure of a system with respect to the values of several metrics and the number of violations of object-oriented design principles. User input is only necessary for setting initial parameters. Our approach suggests a list of behavior preserving refactorings. Of course, the final decision whether to apply or not a proposed refactoring is left to the user, since his understanding of good design might contradict the metrics used as part of the fitness function.

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

Gabriele Bavota[1], Sebastiano Panichella[1], Nikolaos Tsantalis[2], Massimiliano Di Penta[1], Rocco Oliveto[3], Gerardo Canfora[1]
[1]Department of Engineering, University of Sannio, Benevento, Italy
[2]Department of Computer Science & Software Engineering, Concordia University, Canada
[3]Department of Bioscience and Territory, University of Molise, Pesche (IS), Italy
gbavota@unisannio.it, spanichella@unisannio.it, tsantalis@cse.concordia.ca, dipenta@unisannio.it, rocco.oliveto@unimol.it, canfora@unisannio.it

ABSTRACT

Refactoring aims at restructuring existing source code when undisciplined development activities have deteriorated its comprehensibility and maintainability. There exist various approaches for suggesting refactoring opportunities, based on different sources of information, e.g., structural, semantic, and historical. In this paper we claim that an additional source of information for identifying refactoring opportunities, sometimes orthogonal to the ones mentioned above, is team development activity. When the activity of a team working on common modules is not aligned with the current design structure of a system, it would be possible to recommend appropriate refactoring operations—e.g., extract class/method/package—to adjust the design according to the teams' activity patterns. Results of a preliminary study—conducted in the context of extract class refactoring—show the feasibility of the approach, and also suggest that this new refactoring dimension can be complemented with others to build better refactoring recommendation tools.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Documentation, Enhancement, Restructuring, Reverse Engineering, and Reengineering*

Keywords

Refactoring; Developers; Teams

1. INTRODUCTION

Software refactoring is *"a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior"* [7]. Refactoring is usually a direct consequence of an undisciplined evolution, due to the lack of a rigorous and documented development process, poor design decisions, or simply to the need for applying urgent patches to the software.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ASE'14, September 15-19, 2014, Vasteras, Sweden.
Copyright 2014 ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2642948.

In recent and past years, different approaches and tools have been developed to identify refactoring opportunities. Some of them [14] use structural information to identify the need for refactoring, and then suggest a suitable refactoring action. Others also exploit semantic information (i.e., textual information extracted from source code indicating the implementation of similar responsibilities) [2], or historical data [11] to identify refactoring solutions.

We conjecture that a fourth dimension can be highly beneficial when recommending refactoring operations. Such a dimension relates to *changes performed by teams* and it is based on the assumption of congruence between social and technical activities [4]. We define a team as a group of developers that, during the project history, worked on common sets of source code entities (note that in this definition we do not necessarily assume that such developers communicate with each other). The basic idea is that *code entities frequently modified by the same team should be grouped together in a separate module*. A straight-forward consequence of that refactoring would be easing the release of individual components of the project and/or integration activities, without waiting for other teams to complete their tasks.

Suppose that two different teams T_i and T_j usually work on different code entities, and thus they are responsible for maintaining and evolving different features of the system. However, both T_i and T_j work on class C, but on different parts of it, e.g., T_i works on members C_{m_i}, while T_j on members C_{m_j}, where $C_{m_i} \cap C_{m_j} = \varnothing$. This could be a symptom of heterogeneous responsibilities implemented in C, and thus of an opportunity to perform Extract Class refactoring. We claim that a better source code organization can be obtained by separating C_{m_i} from C_{m_j}, because these distinct member groups of C can be likely related to the features being maintained by T_i and T_j, respectively. Note that, as it will be clearer later in the paper, relying solely in change history is insufficient. This is because history would just identify co-changes, while in this context we need to identify code entities co-maintained by the same team members.

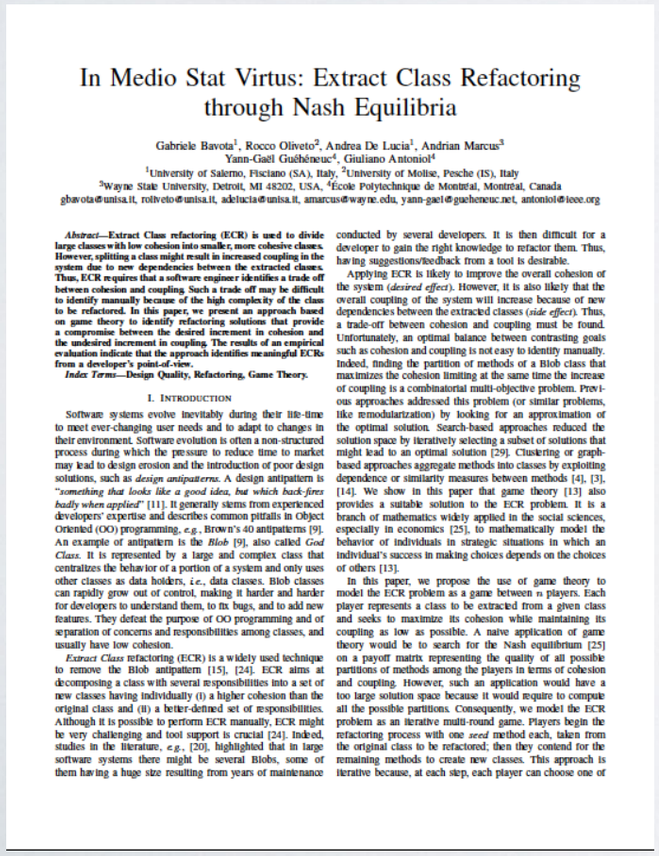
In this paper we (i) introduce the approach named Team Based Refactoring (TBR), to identify refactoring opportunities based on team co-maintenance patterns, (ii) instantiate TBR to support Extract Class refactoring, and (iii) provide preliminary results showing that TBR is able to identify meaningful refactoring solutions and complementary in many cases to other sources of information. This opens the road towards better refactoring recommendation tools being able to provide more accurate and/or more complete suggestions by combining multiple sources of information.

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

Gabriele Bavota[1], Rocco Oliveto[2], Andrea De Lucia[1], Andrian Marcus[3]
Yann-Gaël Guéhéneuc[4], Giuliano Antoniol[4]
[1]University of Salerno, Fisciano (SA), Italy, [2]University of Molise, Pesche (IS), Italy
[3]Wayne State University, Detroit, MI 48202, USA, [4]École Polytechnique de Montréal, Montréal, Canada
gbavota@unisa.it, roliveto@unisa.it, adelucia@unisa.it, amarcus@wayne.edu, yann-gael@gueheneuc.net, antoniol@ieee.org

***Abstract*—Extract Class refactoring (ECR) is used to divide large classes with low cohesion into smaller, more cohesive classes. However, splitting a class might result in increased coupling in the system due to new dependencies between the extracted classes. Thus, ECR requires that a software engineer identifies a trade off between cohesion and coupling. Such a trade off may be difficult to identify manually because of the high complexity of the class to be refactored. In this paper, we present an approach based on game theory to identify refactoring solutions that provide a compromise between the desired increment in cohesion and the undesired increment in coupling. The results of an empirical evaluation indicate that the approach identifies meaningful ECRs from a developer's point-of-view.**

***Index Terms*—Design Quality, Refactoring, Game Theory.**

I. INTRODUCTION

Software systems evolve inevitably during their life-time to meet ever-changing user needs and to adapt to changes in their environment. Software evolution is often a non-structured process during which the pressure to reduce time to market may lead to design erosion and the introduction of poor design solutions, such as *design antipatterns*. A design antipattern is *"something that looks like a good idea, but which backfires badly when applied"* [11]. It generally stems from experienced developers' expertise and describes common pitfalls in Object Oriented (OO) programming, *e.g.*, Brown's 40 antipatterns [9]. An example of antipattern is the *Blob* [9], also called *God Class*. It is represented by 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. 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. They defeat the purpose of OO programming and of separation of concerns and responsibilities among classes, and usually have low cohesion.

Extract Class refactoring (ECR) is a widely used technique to remove the Blob antipattern [15], [24]. ECR aims at decomposing a class with several responsibilities into a set of new classes having individually (i) a higher cohesion than the original class and (ii) a better-defined set of responsibilities. Although it is possible to perform ECR manually, ECR might be very challenging and tool support is crucial [24]. Indeed, studies in the literature, *e.g.*, [20], highlighted that in large software systems there might be several Blobs, some of them having a huge size resulting from years of maintenance conducted by several developers. It is then difficult for a developer to gain the right knowledge to refactor them. Thus, having suggestions/feedback from a tool is desirable.

Applying ECR is likely to improve the overall cohesion of the system (*desired effect*). However, it is also likely that the overall coupling of the system will increase because of new dependencies between the extracted classes (*side effect*). Thus, a trade-off between cohesion and coupling must be found. Unfortunately, an optimal balance between contrasting goals such as cohesion and coupling is not easy to identify manually. Indeed, finding the partition of methods of a Blob class that maximizes the cohesion limiting at the same time the increase of coupling is a combinatorial multi-objective problem. Previous approaches addressed this problem (or similar problems, like remodularization) by looking for an approximation of the optimal solution. Search-based approaches reduced the solution space by iteratively selecting a subset of solutions that might lead to an optimal solution [29]. Clustering or graph-based approaches aggregate methods into classes by exploiting dependence or similarity measures between methods [4], [3], [14]. We show in this paper that game theory [13] also provides a suitable solution to the ECR problem. It is a branch of mathematics widely applied in the social sciences, especially in economics [25], to mathematically model the behavior of individuals in strategic situations in which an individual's success in making choices depends on the choices of others [13].

In this paper, we propose the use of game theory to model the ECR problem as a game between n players. Each player represents a class to be extracted from a given class and seeks to maximize its cohesion while maintaining its coupling as low as possible. A naive application of game theory would be to search for the Nash equilibrium [25] on a payoff matrix representing the quality of all possible partitions of methods among the players in terms of cohesion and coupling. However, such an application would have a too large solution space because it would require to compute all the possible partitions. Consequently, we model the ECR problem as an iterative multi-round game. Players begin the refactoring process with one *seed* method each, taken from the original class to be refactored; then they contend for the remaining methods to create new classes. This approach is iterative because, at each step, each player can choose one of

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

Empir Software Eng (2013) 18:901–932
DOI 10.1007/s10664-012-9226-8

Using structural and semantic measures to improve software modularization

Gabriele Bavota · Andrea De Lucia · Andrian Marcus · Rocco Oliveto

Published online: 14 September 2012
© Springer Science+Business Media, LLC 2012
Editors: Giuliano Antoniol and Martin Pinzger

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

G. Bavota · A. De Lucia
University of Salerno, Fisciano SA, Italy
G. Bavota
e-mail: gbavota@unisa.it
A. De Lucia
e-mail: adelucia@unisa.it
A. Marcus
Wayne State University, Detroit, MI 48202, USA
e-mail: amarcus@wayne.edu
R. Oliveto (✉)
University of Molise, Pesche IS, Italy
e-mail: rocco.oliveto@unimol.it

Springer

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

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.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© YYYY ACM 1049-331X/YYYY/01-ARTA $10.00
DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

Supported Refactoring Operation: Move Class Refactoring

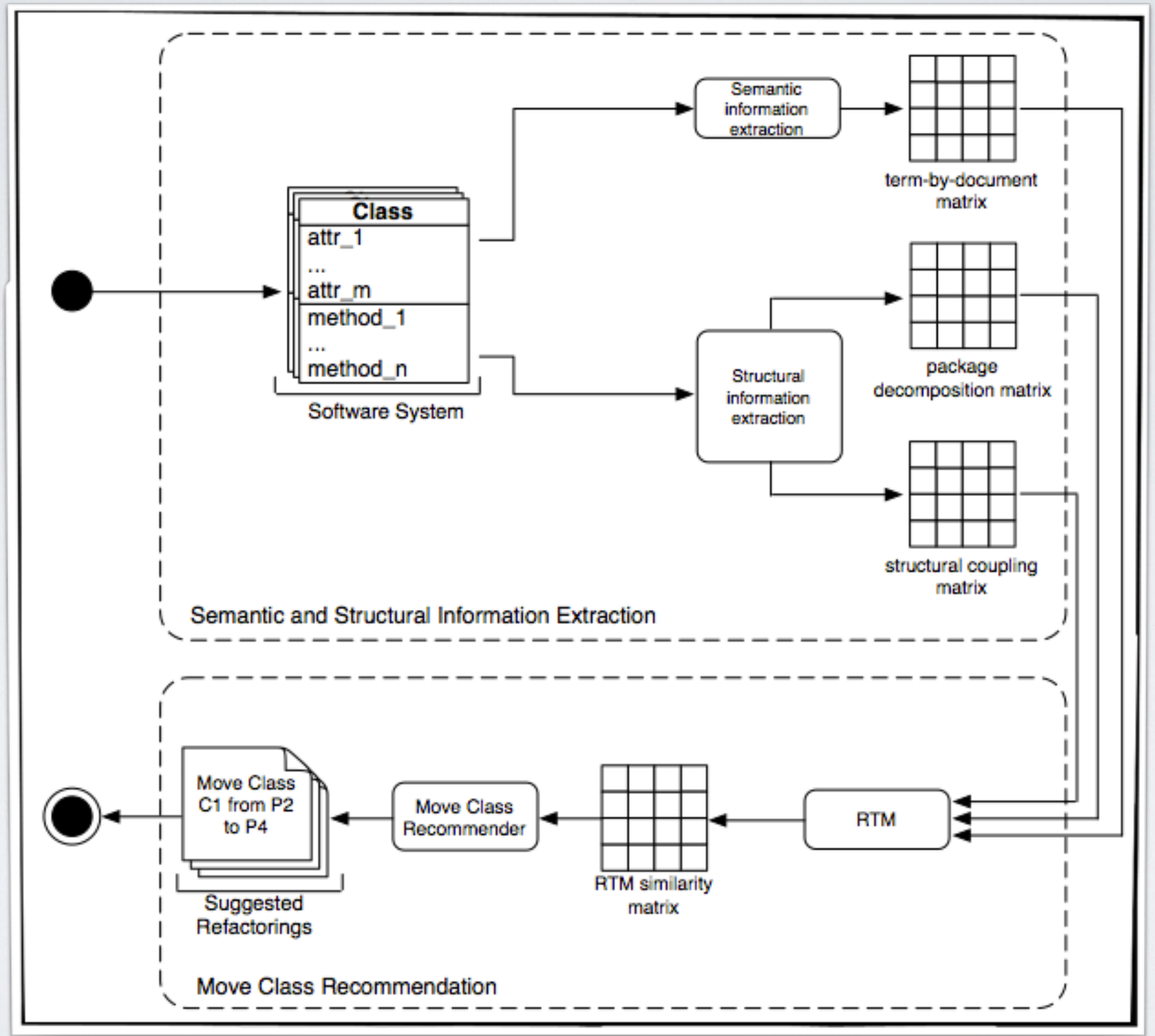
Exploited Information: Method calls, original design, semantic similarity

Algorithm: Heuristic-based

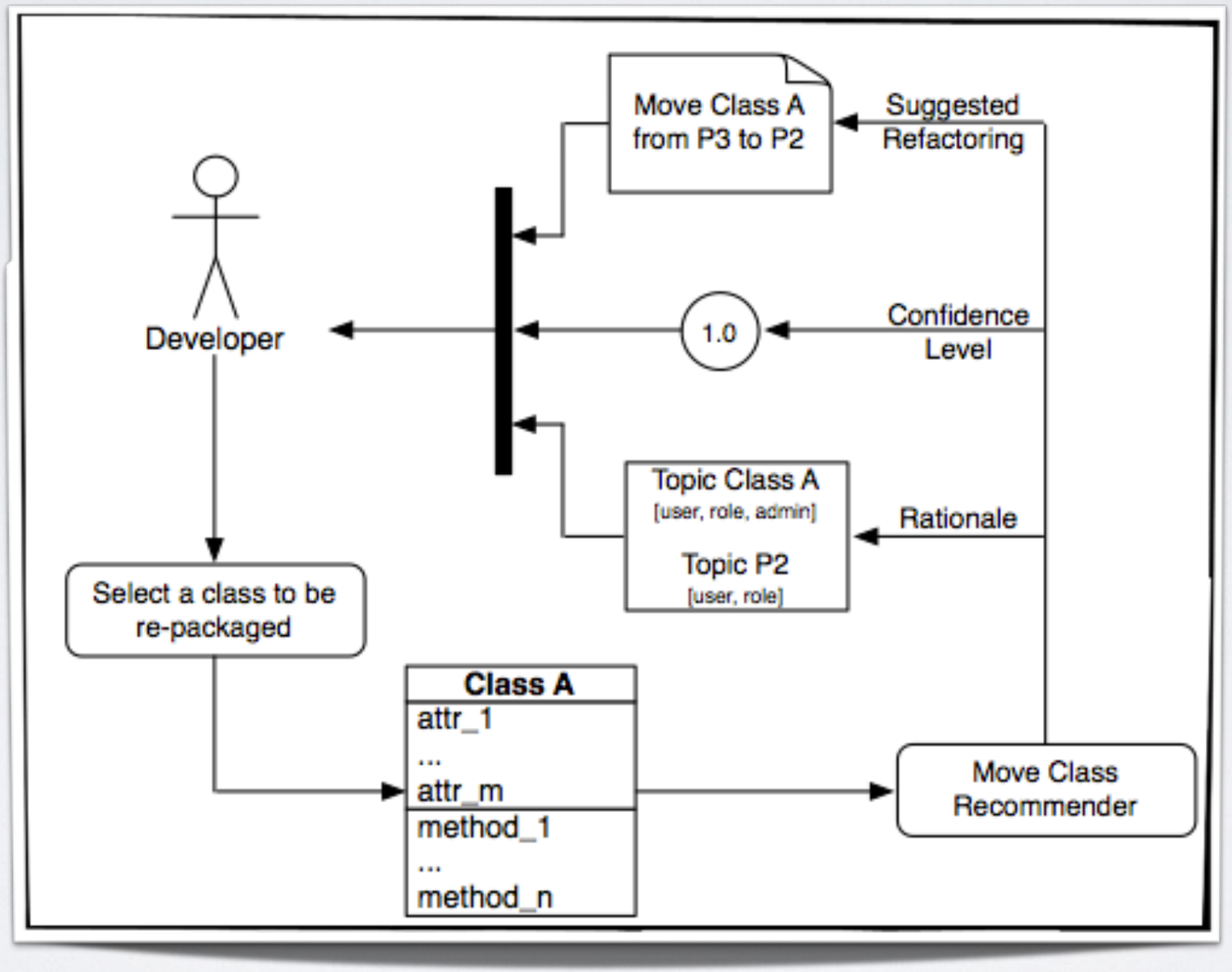
Details in next slides

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]

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 constitute 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: nikos@java.uom.gr, achat@uom.gr.*

Manuscript received 15 Apr. 2008; revised 5 Dec. 2008; accepted 15 Dec. 2008; published online 6 Jan. 2009. Recommended for acceptance by H. Ossher. 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.

0098-5589/09/$25.00 © 2009 IEEE Published by the IEEE Computer Society

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

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 40, NO. 7, JULY 2014 671

Methodbook: Recommending Move Method Refactorings via Relational Topic Models

Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia

Abstract—During software maintenance and evolution the internal structure of the software system undergoes continuous changes. These modifications drift the source code away from its original design, thus deteriorating its quality, including cohesion and coupling of classes. Several refactoring methods have been proposed to overcome this problem. In this paper we propose a novel technique to identify *Move Method* refactoring opportunities and remove the *Feature Envy* bad smell from source code. Our approach, coined as Methodbook, is based on relational topic models (RTM), a probabilistic technique for representing and modeling topics, documents (in our case methods) and known relationships among these. Methodbook uses RTM to analyze both structural and textual information gleaned from software to better support move method refactoring. We evaluated Methodbook in two case studies. The first study has been executed on six software systems to analyze if the move method operations suggested by Methodbook help to improve the design quality of the systems as captured by quality metrics. The second study has been conducted with eighty developers that evaluated the refactoring recommendations produced by Methodbook. The achieved results indicate that Methodbook provides accurate and meaningful recommendations for move method refactoring operations.

Index Terms—Refactoring, relational topic models, empirical studies

1 Introduction

One of the main goals of software development is to tackle the complexity of software. In object-oriented (OO) software, classes are the primary decomposition mechanism, which group together data and operations to reduce complexity. Researchers defined coupling and cohesion as properties of software modules, like class and packages. In particular, coupling has been defined as *the degree to which each module relies on each one of the other modules* while cohesion is *the degree to which the elements of a module belong together* [1]. Generally, accepted rules state that modules should have high cohesion and low coupling [1], [2], [3]. In fact, several empirical studies provided evidence that high levels of coupling and/or lack of cohesion are generally associated with lower productivity, greater rework, and more significant design efforts by developers (e.g., [4], [5]). In addition, classes with low cohesion and/or high coupling have been shown to correlate with high defect rates [6].

Software systems inevitably evolve during their lifecycle to meet ever-changing users' needs and adapt to changes in their environment. During evolution, the structural design of the software system changes and the changing forces mostly results in a deterioration of the software structure which also exhibits worse values of cohesion and coupling. Indeed, software evolution is often an unstructured process during which the developers' needs to reduce time to market may lead to design erosion and the introduction of poor design solutions, usually referred to as *code bad smells* [7], [8].

A classic bad small is the *Feature Envy*, arising when a method seems to be more interested in a class other than the one it is implemented in [8]. For example, instances of this smell are present when a method invokes many times methods of another class (i.e., the envied class) or, more in general, when the responsibilities it implements are more similar to those grouped in the envied class than to those of the class it is implemented in. This clearly results in reduced class cohesion and increased coupling between classes. Thus, it is important to identify and remove the *Feature Envy* bad smell whenever instances are found in a software system. In order to remove such a bad smell, a *Move Method* refactoring operation is required, i.e., the method is moved to the envied class. Unfortunately, not all the cases are cut-and-dried. Often a method uses features of several classes, thus the identification of the envied class (as well as the method to be moved) is not always trivial [9].

These considerations highlight the need for automated support to assist developers in making adequate decisions while analyzing constantly changing structural and textual information in evolving software. In this paper, we present an approach to identify Move Method refactoring opportunities aimed at solving Feature Envy bad smell. The proposed approach, named Methodbook, follows the Facebook[1] metaphor. Facebook is a well-known

• *G. Bavota is with the University of Sannio, Benevento, Italy. E-mail: gbavota@unisannio.it.*
• *R. Oliveto is with the University of Molise, Pesche (IS), Italy. E-mail: rocco.oliveto@unimol.it.*
• *M. Gethers is with the Information Systems Department, University of Maryland, Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250. E-mail: mgethers@umbc.edu.*
• *D. Poshyvanyk is with The College of William and Mary, McGlothlin-Street Hall 006, Williamsburg, VA 23185. E-mail: denys@cs.wm.edu.*
• *A. De Lucia is with the University of Salerno, Fisciano (SA), Italy. E-mail: adelucia@unisa.it.*

Manuscript received 29 Jan. 2013; revised 22 Aug. 2013; accepted 19 Nov. 2013. Date of publication 15 Dec. 2013; date of current version 18 July 2014. Recommended for acceptance by M. Robillard. 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.2013.60

1. http://www.facebook.com/.

0098-5589 © 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

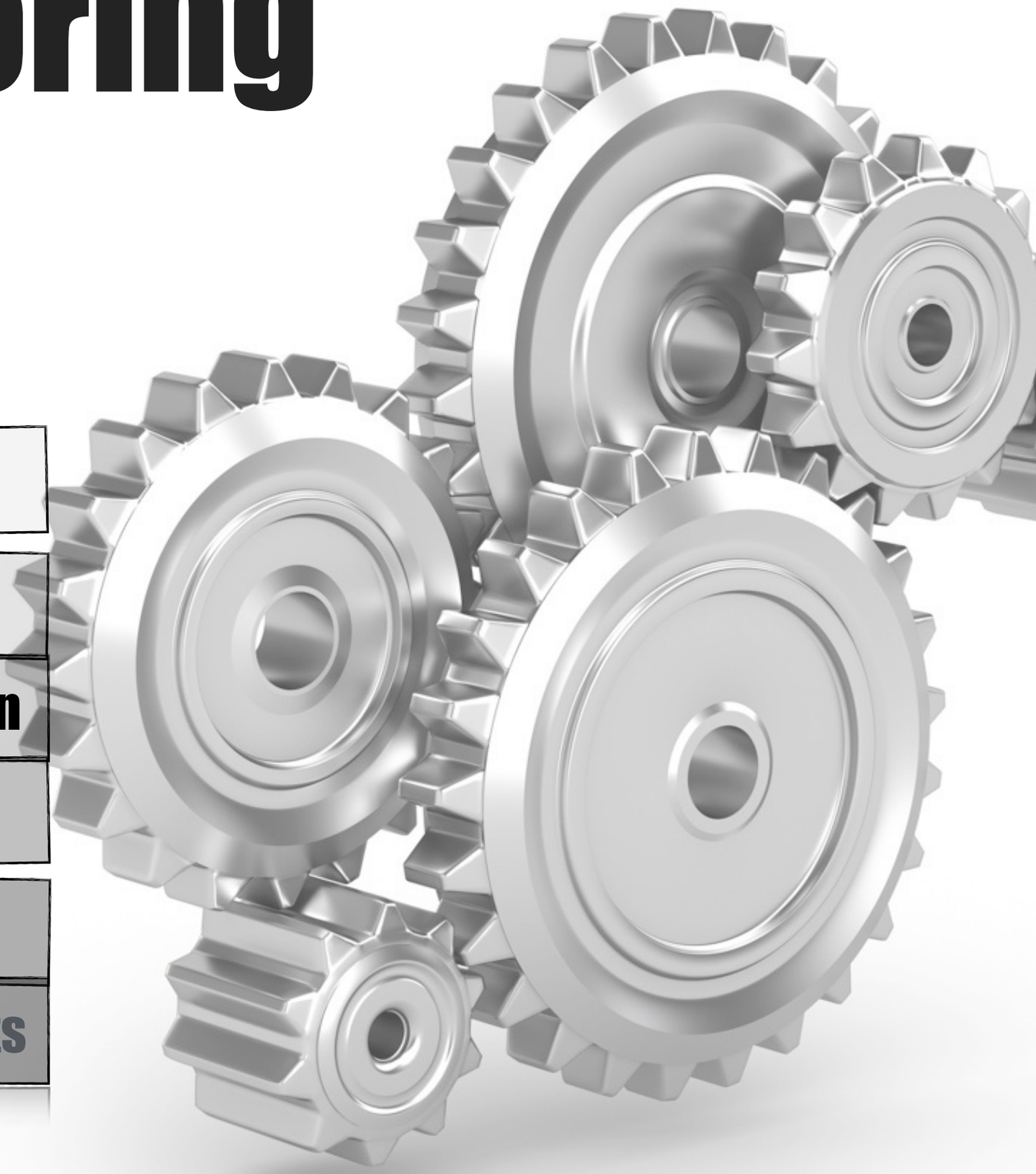
Supported Refactoring Operation: Move Method Refactoring

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.
Copyright 2007 ACM 978-1-59593-811-4/07/0009 ...$5.00.

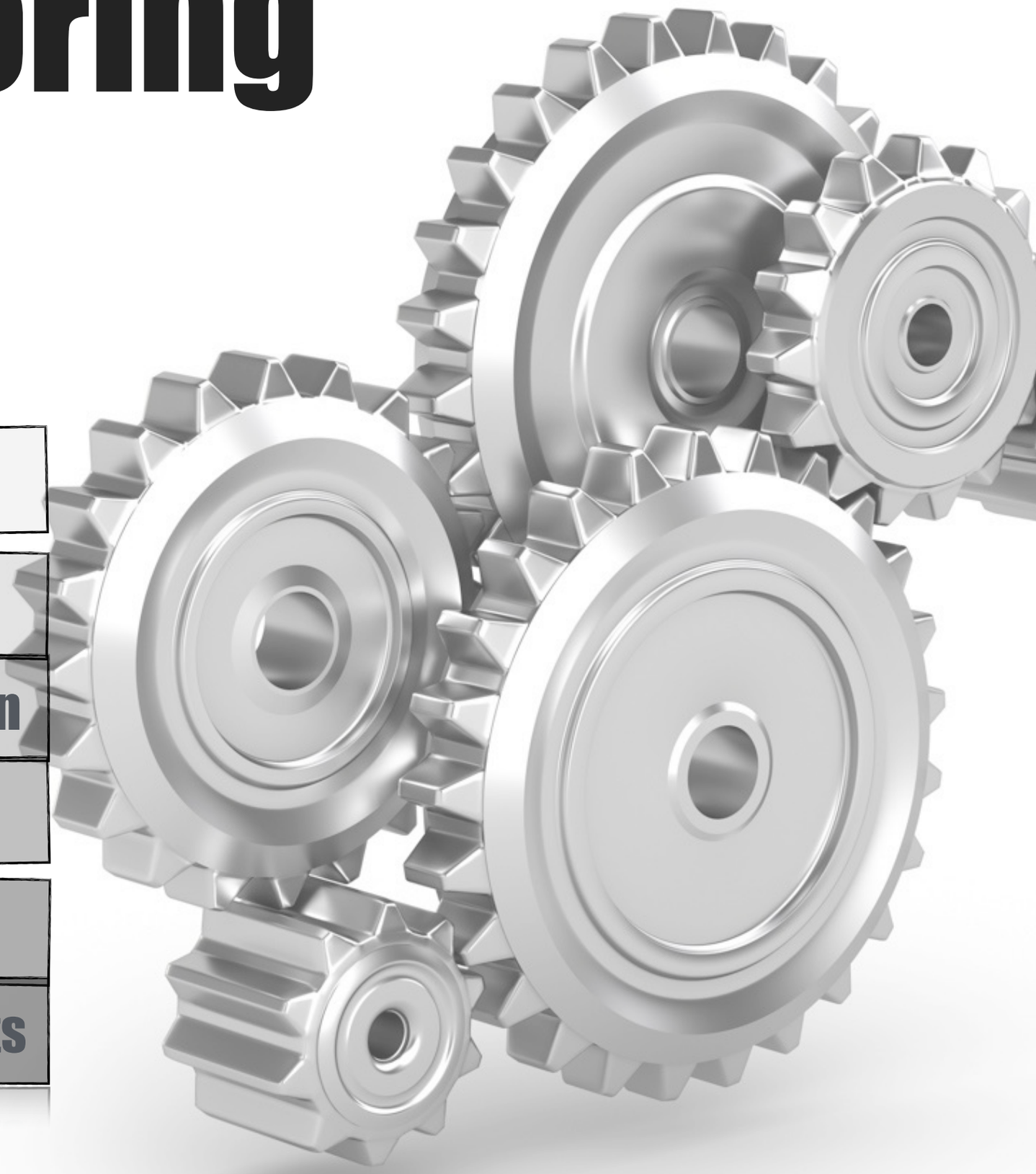
Even automated refactoring performed by Integrated Development Environments could be fault-prone

45 previously unreported bugs in Eclipse and NetBeans

[Daniel et al. ESEC/FSE 2007]

No approaches fully tested, implemented in tools, able to support a wide range of refactoring operations

The refactoring process



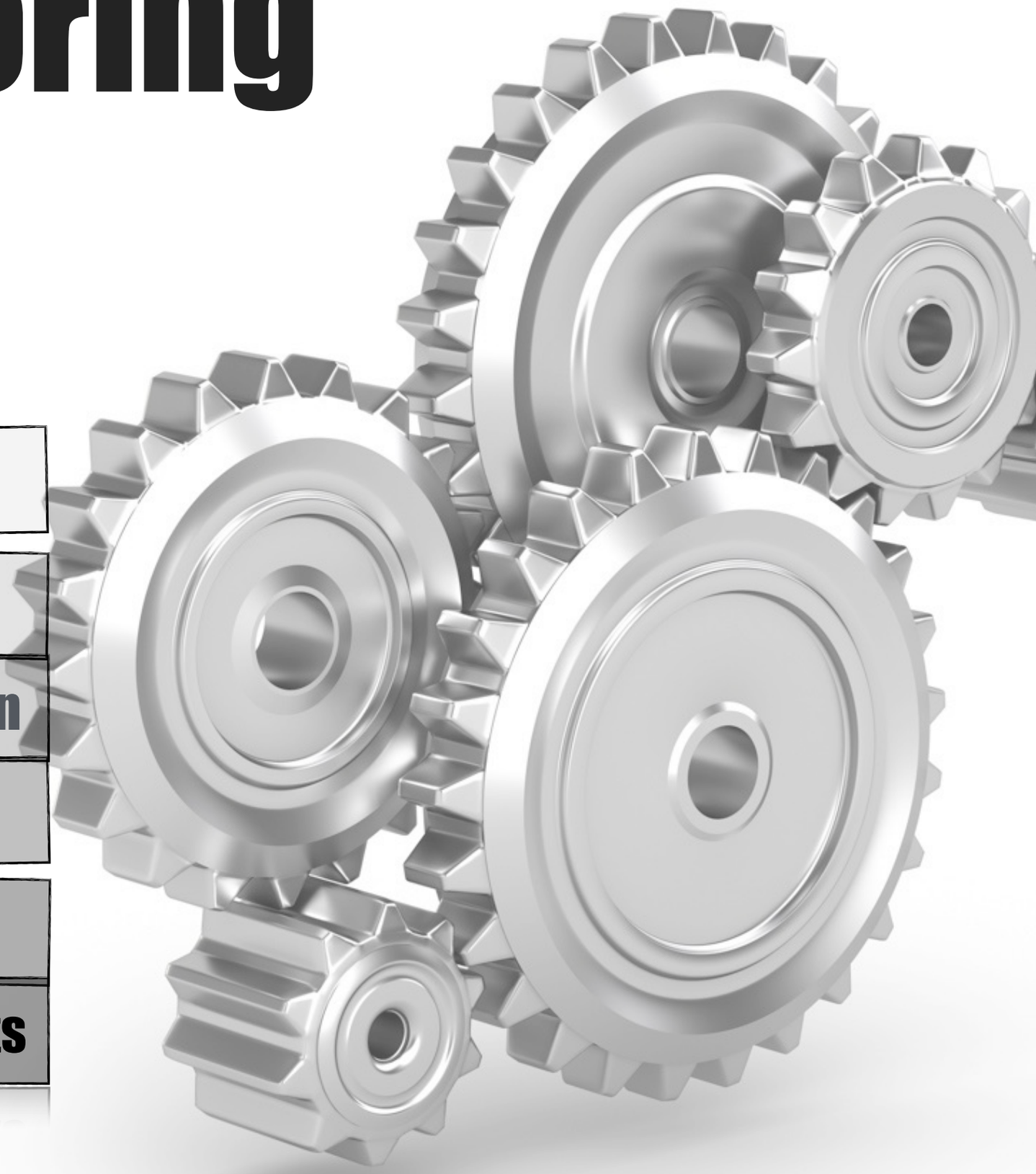
- Where to refactor
- How to refactor?
- Guarantee behaviour preservation
- Apply the refactoring
- Assess its effects on quality
- Consistently modify other artifacts

[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

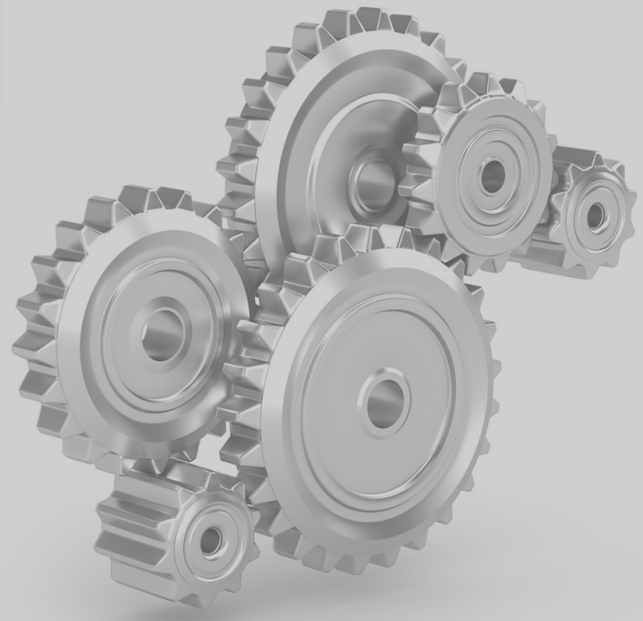
Recovering traceability when needed, see e.g., Quesef et al. “Recovering test-to-code traceability using slicing and textual analysis”, JSS 2014

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

Part I

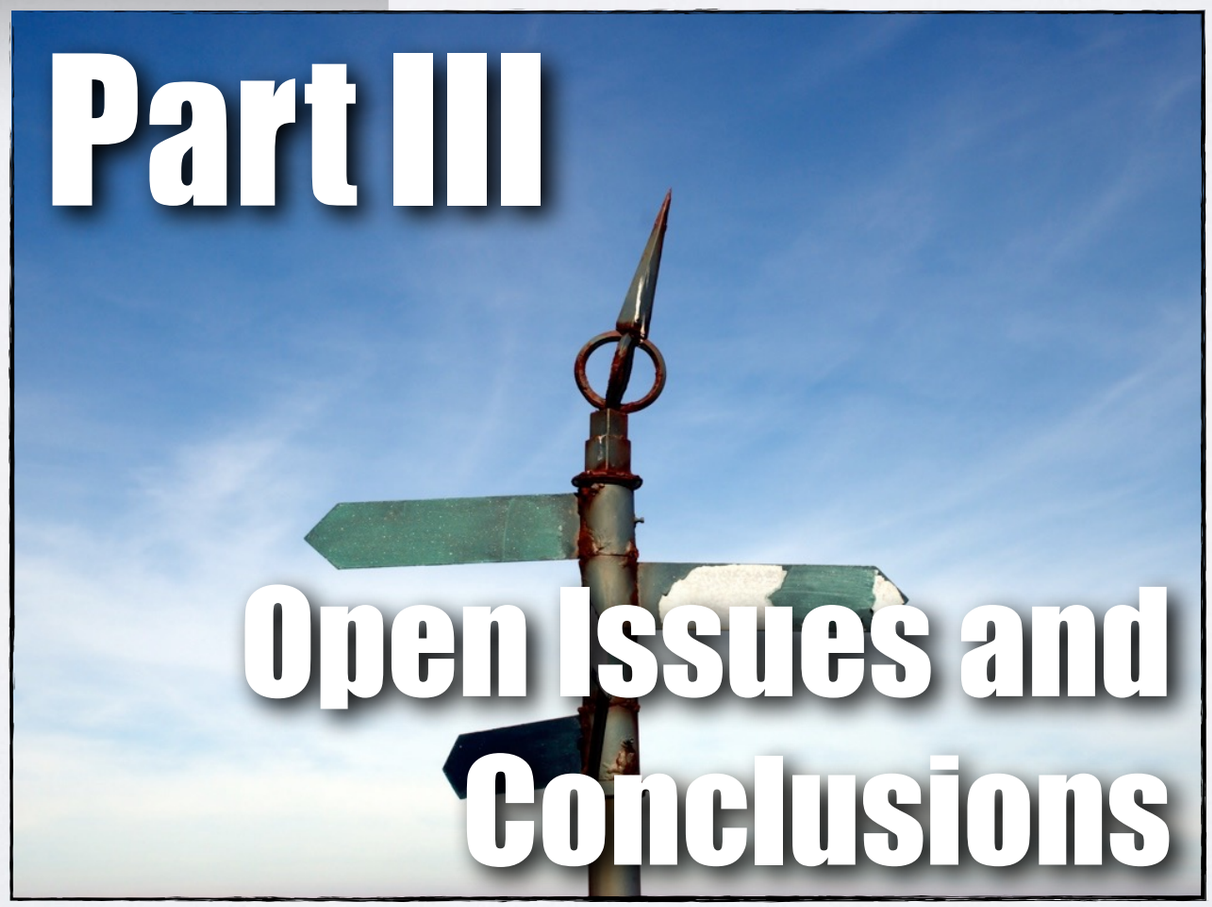
Bad Code Smells and Software Refactoring

Part II



The refactoring process

Part III



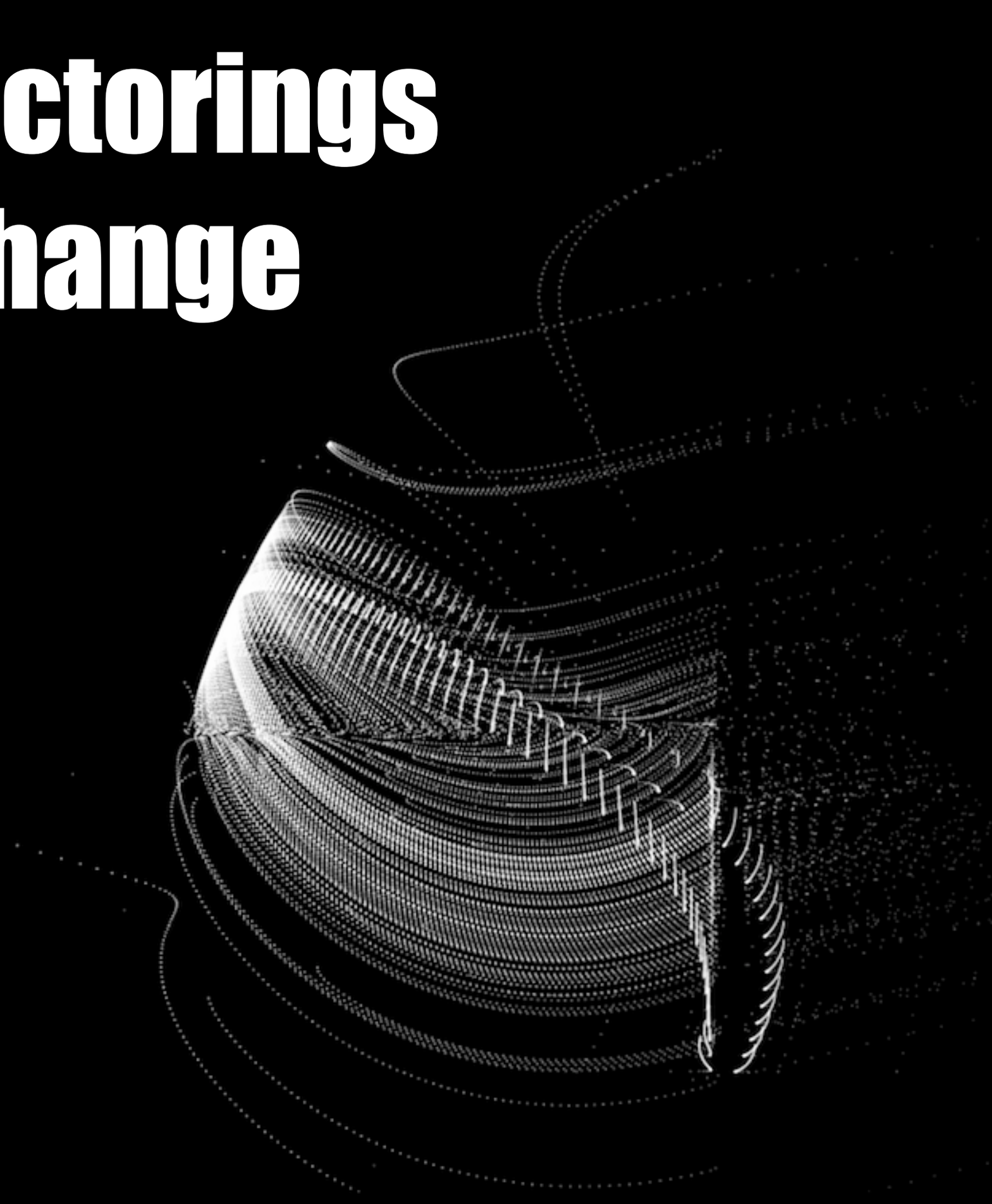
Open Issues and Conclusions

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
© Springer Science+Business Media, LLC 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
© Springer Science+Business Media, LLC 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

Springer

[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, oraziostrollo@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 resources/skills, 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 [2] 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.

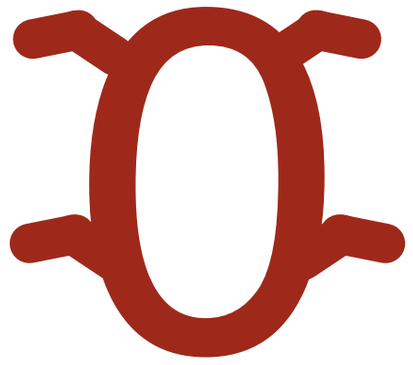
While there are attempts to investigate the relation between some refactorings and fault-proneness [8], [9] or change entropy [7], to the best of our knowledge there is no study aimed at thoroughly investigating whether a wide set of (even undocumented) refactorings occurred in a software system during 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/

978-0-7695-4783-1/12 $26.00 © 2012 IEEE
DOI 10.1109/SCAM.2012.20

IEEE computer society



Bavota et al. [SCAM 2012]

63 releases of 3 systems

Apache Ant ArgoUML Xerces

15,008

refactorings of 52 different types

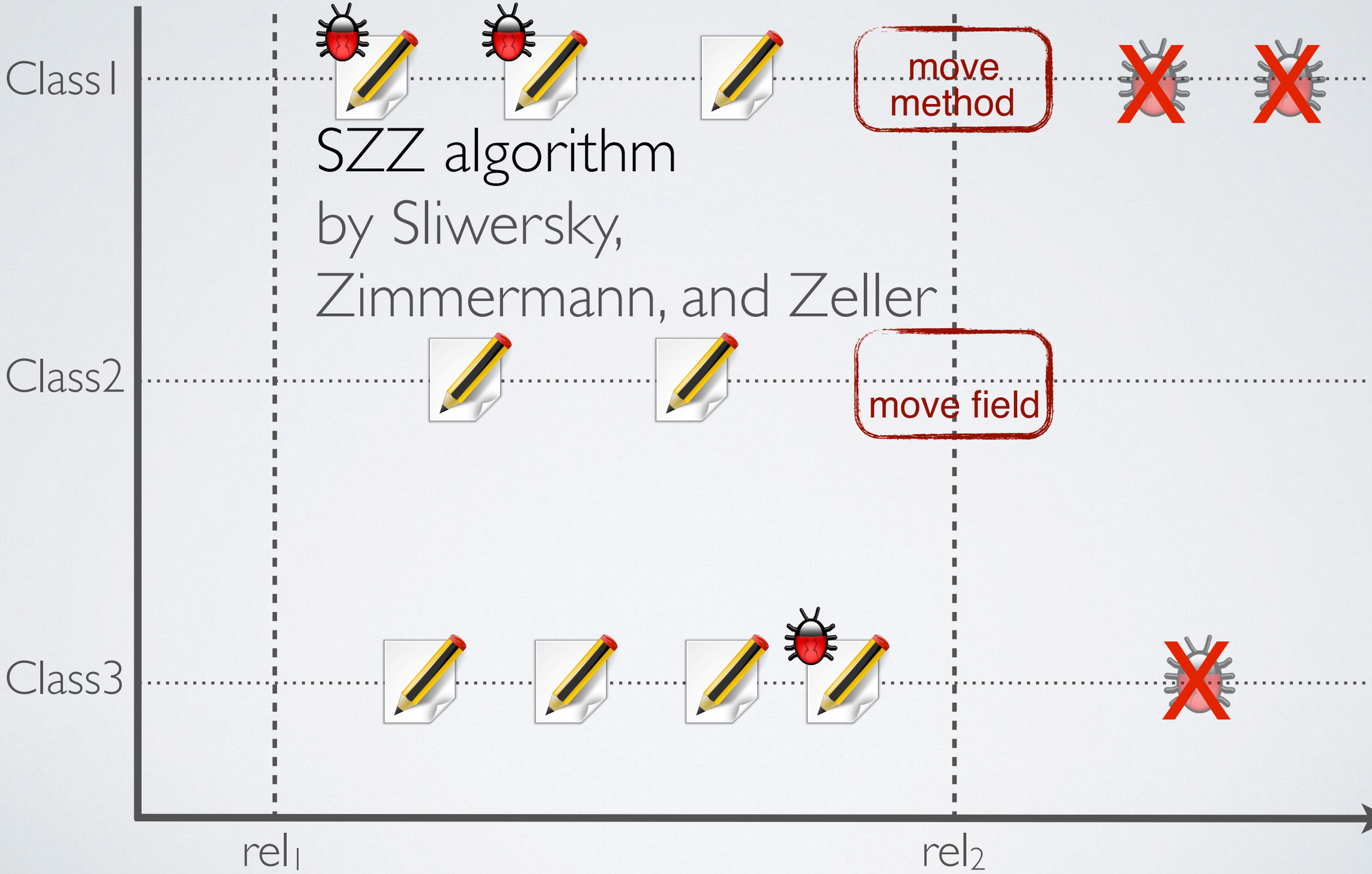
MANUALLY VALIDATED

12,922 refactorings of 52 different types

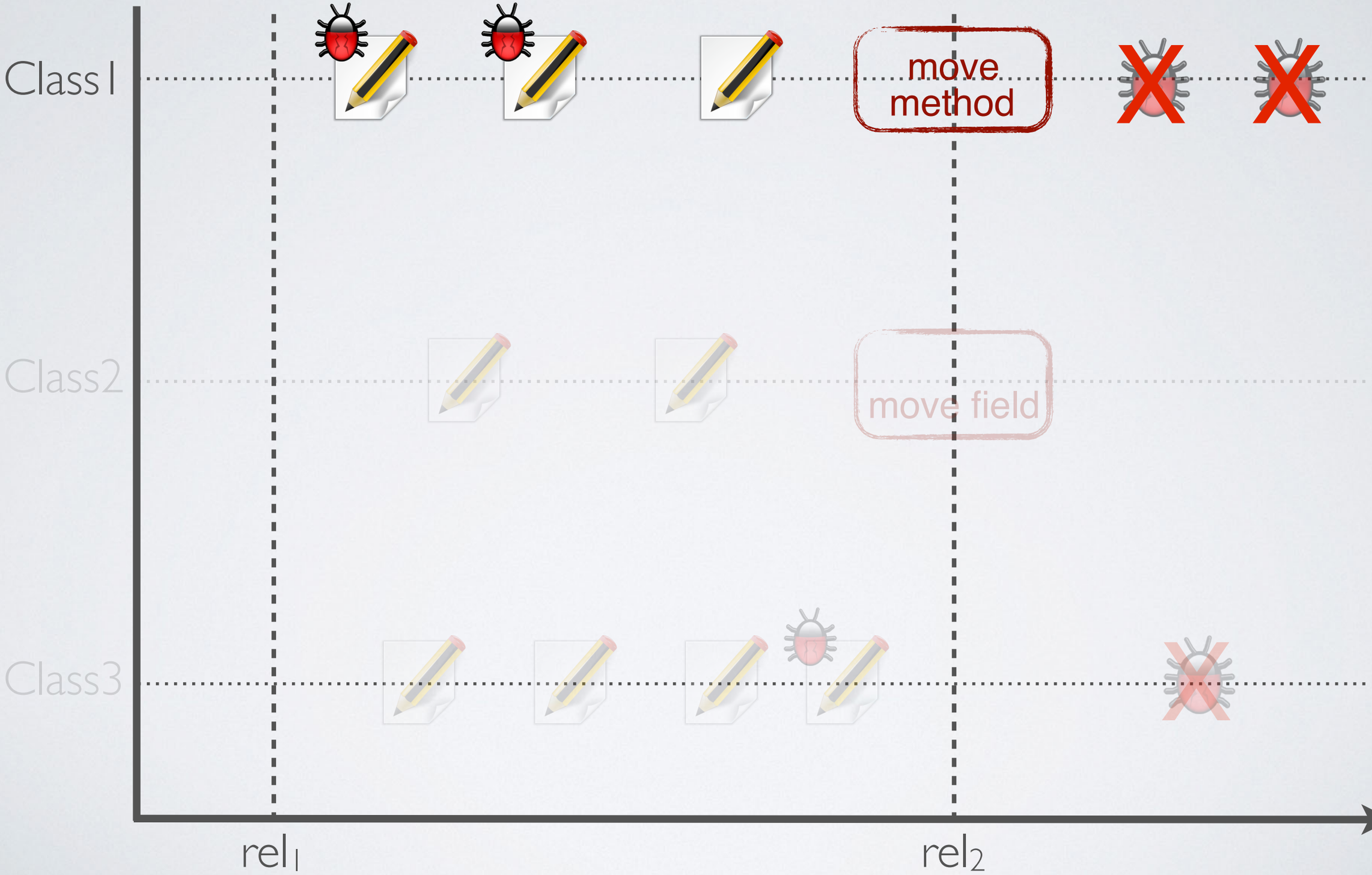
what about bugs?

we considered only fixed bugs

bug-inducing refactorings for rel_2



bug-inducing refactorings for rel_2



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.

© 2015 Elsevier Inc. All rights reserved.

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/© 2015 Elsevier Inc. All rights reserved.

[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 them 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
<u>Technical Debt</u>

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

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

Avoiding Risks

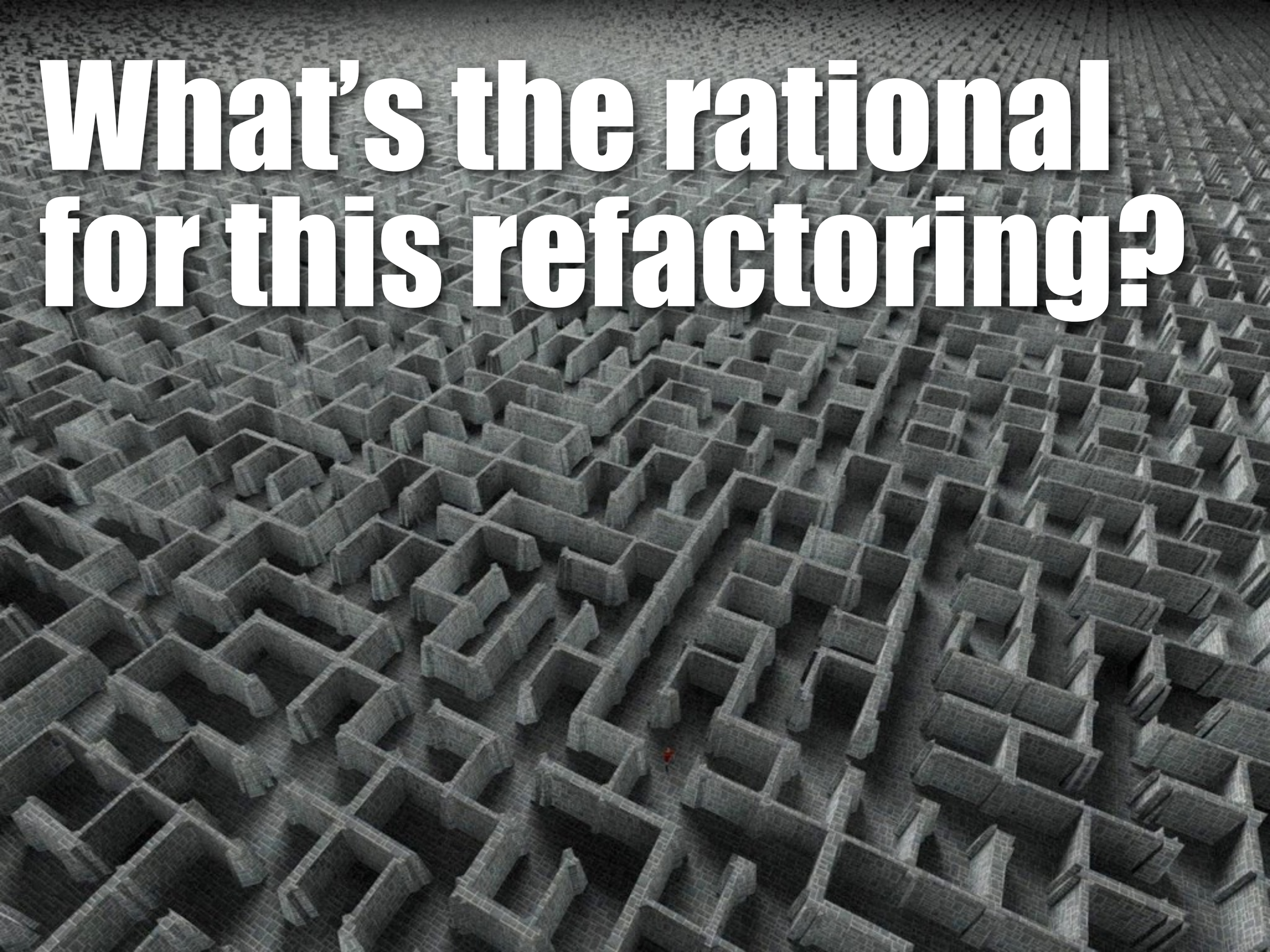
More automated support ...


...preserving
system behavior

Convincing Developers

Usability of refactoring tools

What's 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.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© YYYY ACM 1049-331X/YYYY/01-ARTA $10.00
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.
© Springer-Verlag Berlin Heidelberg 2012

Bavota et al. [SSBSE 2012]

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[2], 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-mails: 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 dectector 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]

Code Smell Detection: New Frontiers

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

Textual Analysis for Code Smell Detection

Fabio Palomba
Department of Management & Information Technology
University of Salerno, Fisciano (SA), Italy
Advisors: Andrea De Lucia, Rocco Oliveto

***Abstract*—The negative impact of smells on the quality of a software systems has been empirical investigated in several studies. This has recalled the need to have approaches for the identification and the removal of smells. While approaches to remove smells have investigated the use of both structural and conceptual information extracted from source code, approaches to identify smells are based on structural information only. In this paper, we bridge the gap analyzing to what extent conceptual information, extracted using textual analysis techniques, can be used to identify smells in source code. The proposed textual-based approach for detecting smells in source code, coined as *TACO* (Textual Analysis for Code smell detectiOn), has been instantiated for detecting the Long Method smell and has been evaluated on three Java open source projects. The results indicate that TACO is able to detect between 50% and 77% of the smell instances with a precision ranging between 63% and 67%. In addition, the results show that TACO identifies smells that are not identified by approaches based on solely structural information.**

I. Research Problem and Motivation

Technical debt is a metaphor used to describe the consequences of poor software design and bad coding. Specifically, the debt represents a piece of code that needs to be rewritten or completed before a particular task can be considered complete [9]. The metaphor explains well the trade-offs between delivering the most appropriate but still immature product, in the shortest time possible [7], [9], [13], [14], [24]. Code smells (shortly *smells*), i.e., symptoms of poor design and implementation choices [11], are one of the most important factors contributing to technical debt. In the past and, most notably, in recent years, several studies investigated the relevance that code smells have for developers [21], [32], the extent to which code smells tend to remain in a software system for long periods of time [2], [8], as well as the side effects of code smells, such as increase in change- and fault-proneness [12] or decrease of software understandability [1] and maintainability [25], [31], [30].

The results achieved in these studies have suggested the need to properly manage smells aiming at improving the quality of a software systems. Thus, several approaches and tools have been proposed for detecting smells [17], [18], [19], [20], [22], [23], [26], [27], [28], and, whenever possible, triggering refactoring operations [5], [4], [27]. While approaches to remove smells have investigated the use of both structural and conceptual information extracted from source code, approaches to identify smells are based on structural information only. Recently, Palomba et al. [22] have also used historical information to identify smell. In the context of their study, the authors obtained that using historical information is possible to identify smell instances that are missed using structural information only. In this paper, we conjecture that also by using conceptual information is possible to identify smell instances that are missed by using other sources of information. In other words, we believe that, as obtained in other software engineer tasks (see e.g., [6], [15], [16]), conceptual properties can provide complementary information to structural properties when identifying smells in source code.

In order to verify our conjecture, we present *TACO* (**T**extual **A**nalysis for **C**ode smell detecti**O**n), a textual-based smell detection approach. TACO has been instantiated for the detection of a specific smell, i.e., *Long Method*. However, the approach can be easily extended to other smells. The choice of *Long Method* is not random, but guided by the idea that such a smell is a perfect candidate to evaluate the benefits of conceptual information. Indeed, a method with a high number of lines of code likely implements different responsibilities and thus textual analysis could be particularly suitable to identify such responsibilities.

II. Approach and Uniqueness

Fowler [11] described the *Long Method* as a method in which there is the implementation of a main functionality together with auxiliary functions that should be managed in different methods. Thus, the key idea behind TACO is that a *Long Method* contains a set of code blocks conceptually unrelated each that should be managed separately.

Figure 1 overviews the main steps of the proposed approach. First, TACO extracts from a method M_i the blocks composing it, applying the technique proposed by Wang et al. [29]. Then, from each block TACO extracts the identifiers and comments cleaning the text from non-relevant words, such as language keywords. Each cleaned block of code is viewed as a document, and for each pair of code block is computed a value of similarity using Latent Semantic Indexing (LSI) [10]. The similarity values between all the possible pairs of blocks are stored in a block similarity matrix, where a generic entry $c_{i,j}$ represent the similarity between the method blocks b_i and b_j. If in the block similarity matrix there is an entry (i.e., similarity between two code blocks) lower than α, then a *Long Method* istance is identified. The parameter α has been empirically evaluated and set to 0.4.

III. Preliminary Evaluation

We evaluate the accuracy of TACO in detecting *Long Method* smell instances in three software systems, namely

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

[Palomba - ICSE SRC 2015]