THE SCENT OF DESIGN SMELLS

SHOULD DEVELOPERS CARE ABOUT IT?

Foutse Khomh
Associate Professor and FRQ/IVADO Research Chair
foutse.khomh@polymtl.ca
@SWATLab

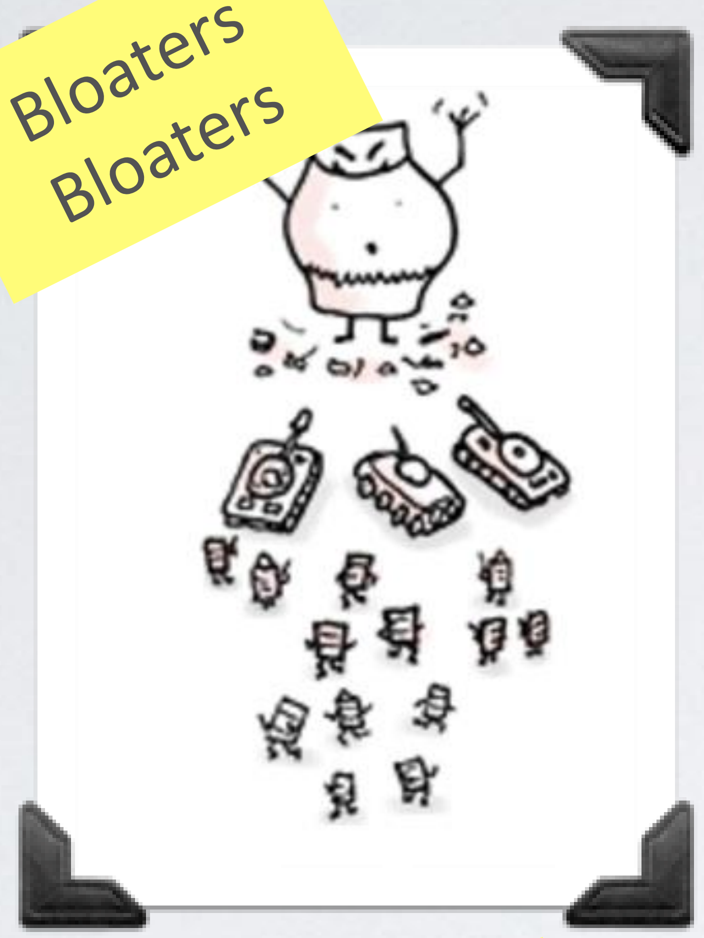
DESIGN DECAY

WHAT ARE DESIGN SMELLS?

“Symptoms of poor design and implementation choices”
[Fowler, 1999]

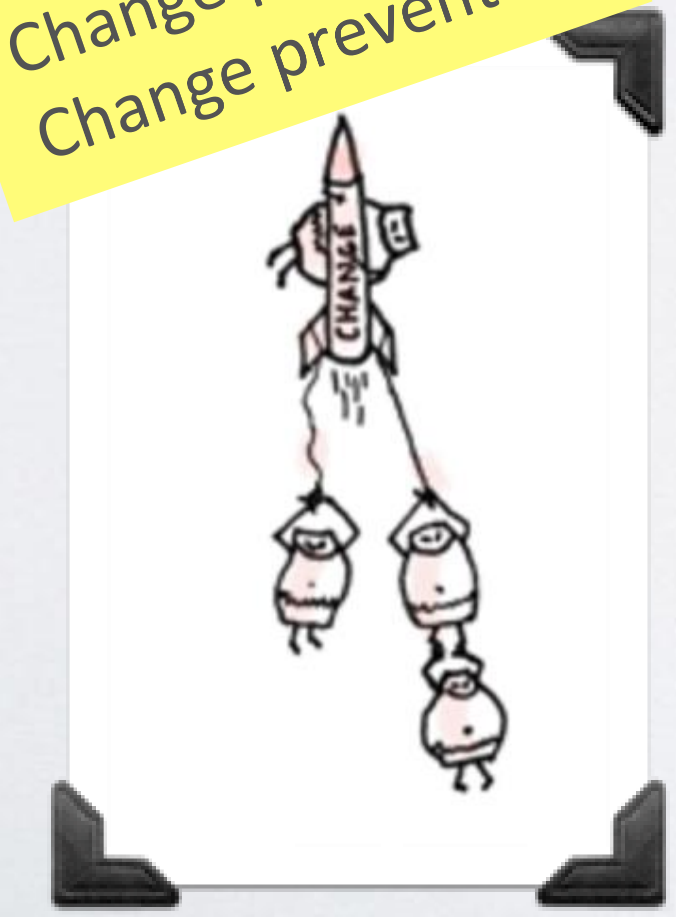
DESIGN SMELLS

Bloaters
Bloaters

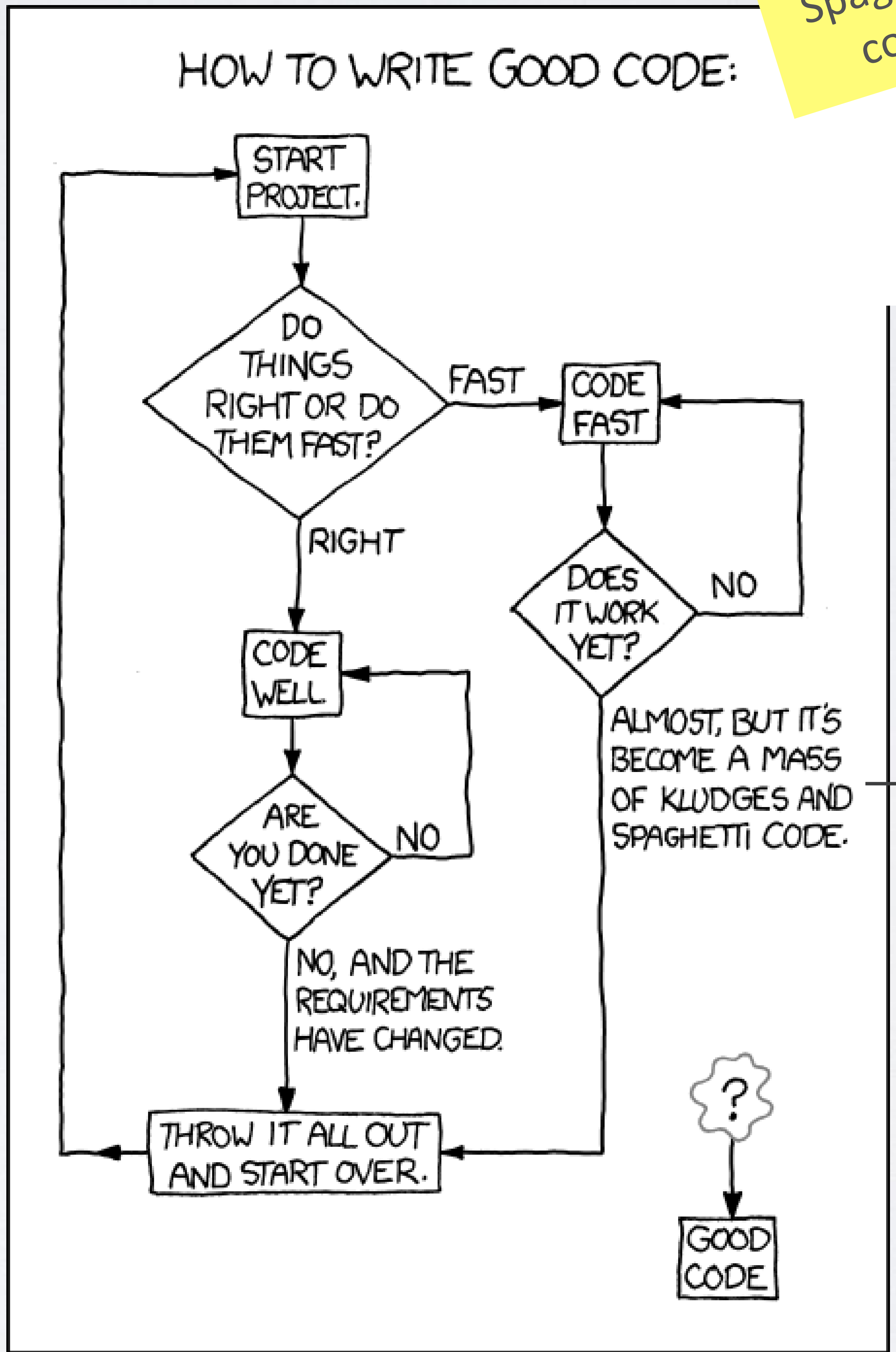


- Long Method
- Large Class

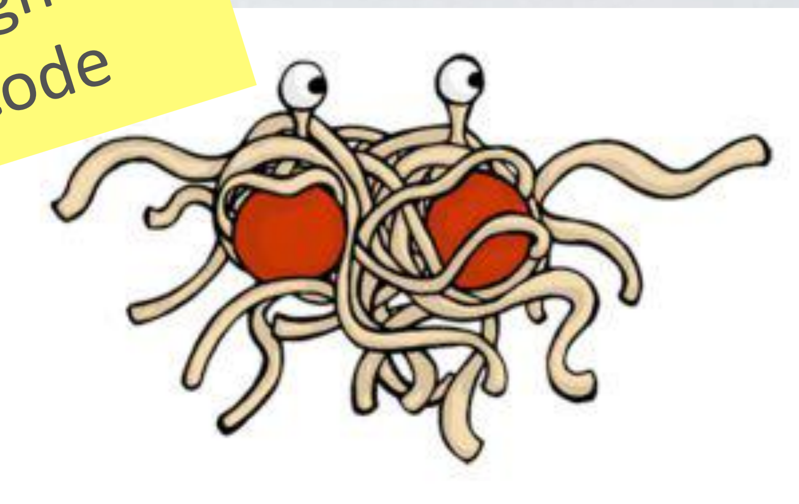
Change preventers
Change preventers



- Divergent Change
- Shotgun Surgery

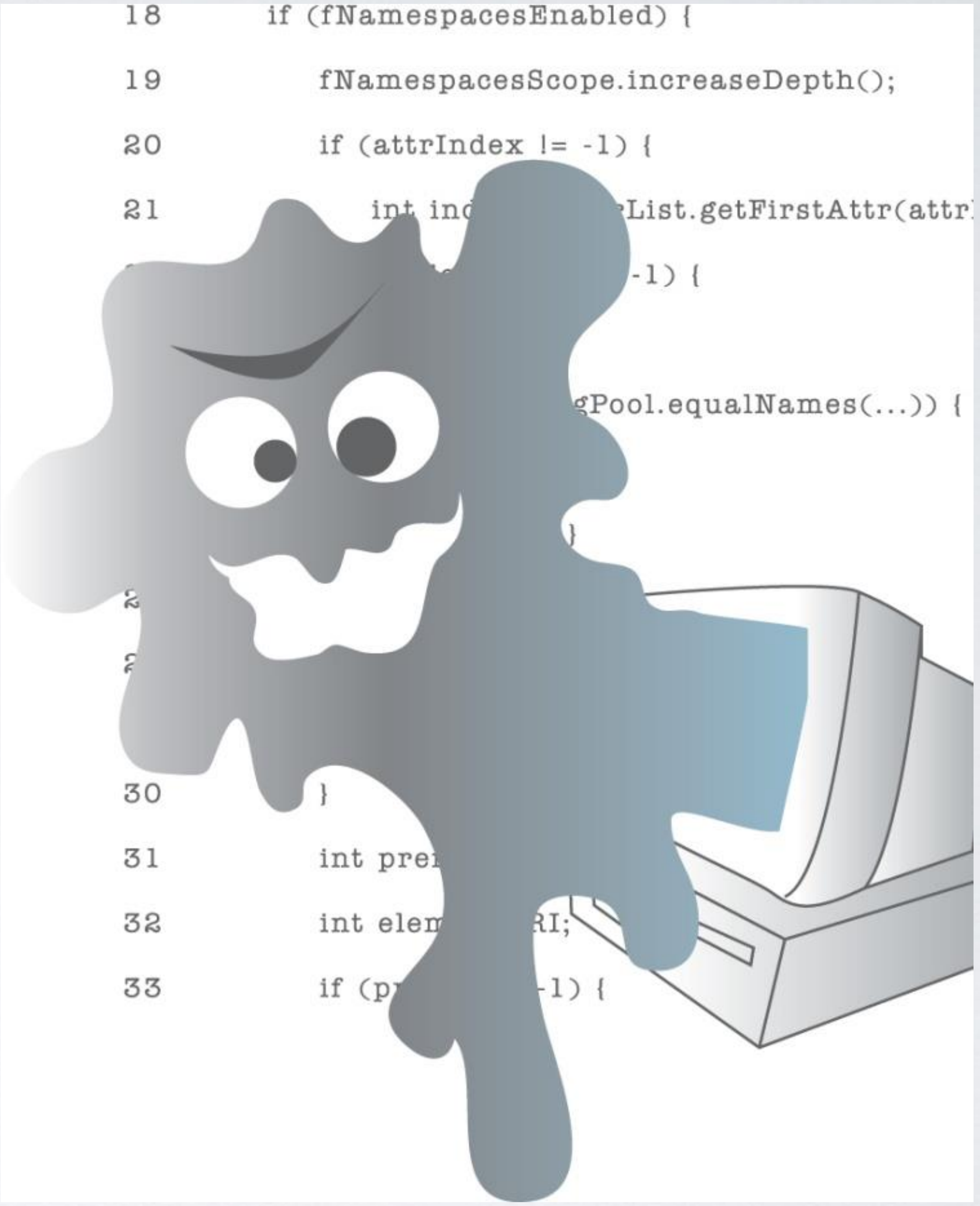


Spaghetti code



BLOB (GOD CLASS)

- **Symptoms:**
  - Large controller class
  - Many fields and methods with a low cohesion*
  - Lack of OO design.
  - Procedural-style than object oriented architectures.



*How closely the methods are related to the instance variables in the class.
Measure: LCOM (Lack of cohesion metric)

BLOB (GOD CLASS)

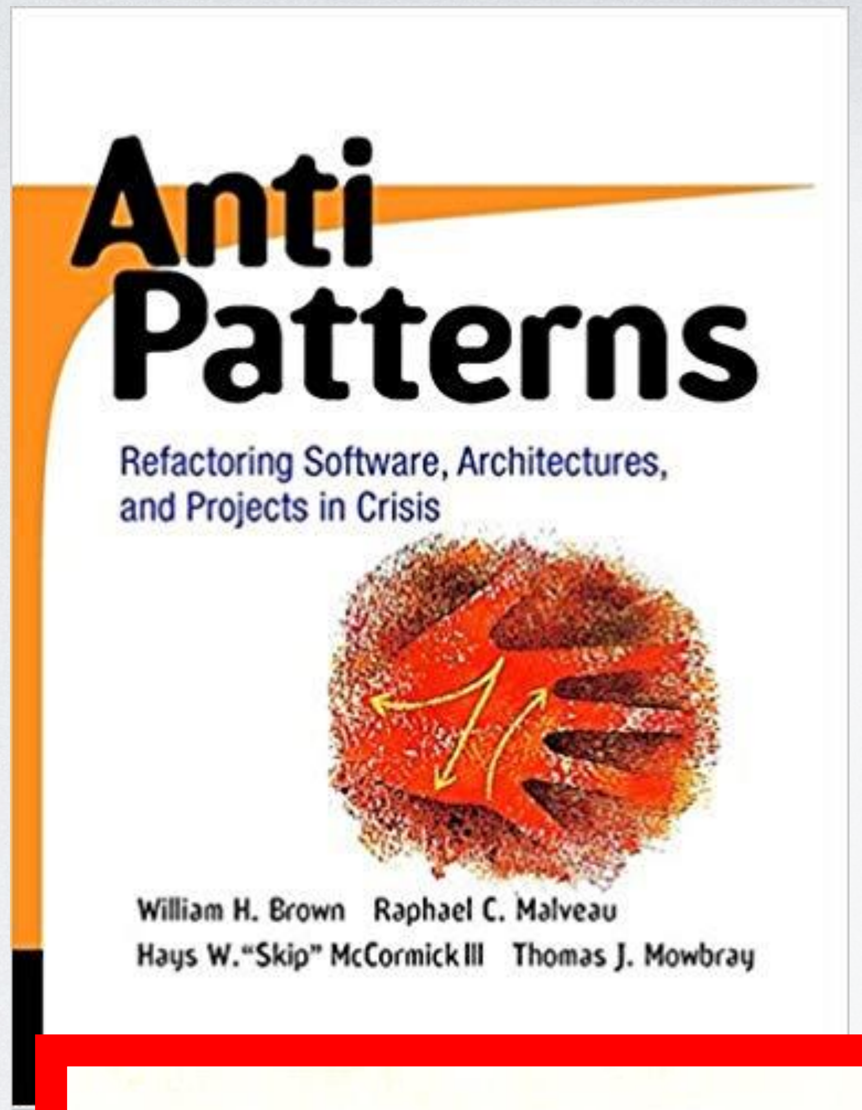
SPECULATIVE GENERALITY

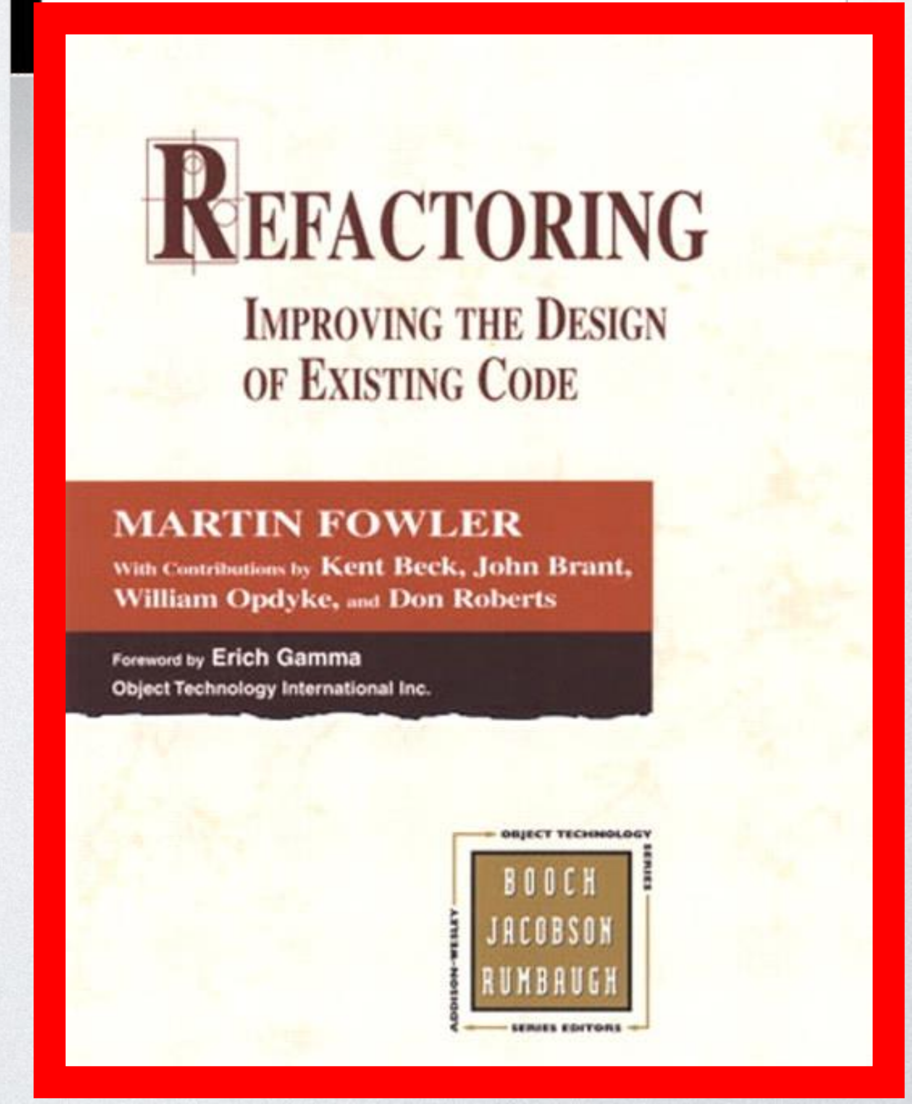
The code is created "just in case" to support anticipated future features that never get implemented

ANTIPATTERNS AND CODE SMELLS CONJECTURED TO…

Impact program comprehension, software evolution and maintenance activities

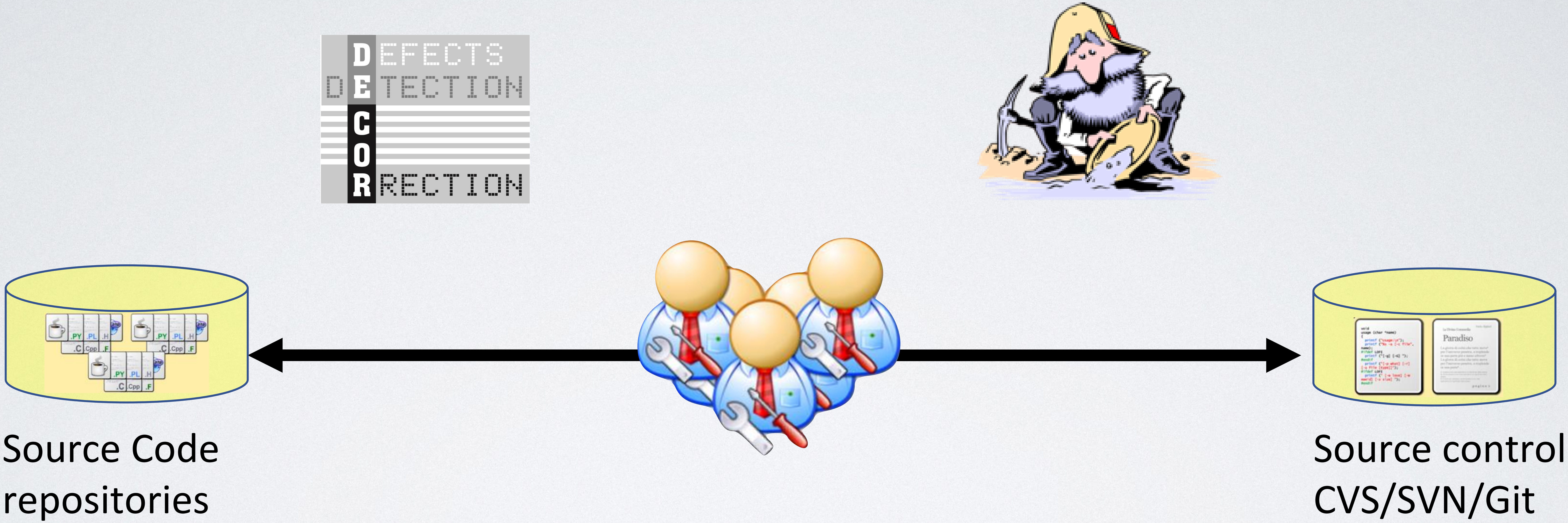
It is important to detect them early in software development process, to reduce the maintenance costs





WHAT WE DID

HOW WE THOUGHT ABOUT IT



INVESTIGATING CHANGE-PRONENESS

(IEEE TCSE MIP Paper)

RELATION BETWEEN CODE SMELLS AND CHANGE-PRONENESS

An Exploratory Study of the Impact of Code Smells on Software Change-proneness

Foutse Khomh[1], Massimiliano Di Penta[2], and Yann-Gaël Guéhéneuc[1]

[1]Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
[2] University of Sannio, Dept. of Engineering, Benevento, Italy

E-mails: {foutsekh, guehene}@iro.umontreal.ca, dipenta@unisannio.it

Abstract

Code smells are poor implementation choices, thought to make object-oriented systems hard to maintain. In this study, we investigate if classes with code smells are more change-prone than classes without smells. Specifically, we test the general hypothesis: classes with code smells are not more change prone than other classes. We detect 29 code smells in 9 releases of Azureus and in 13 releases of Eclipse, and study the relation between classes with these code smells and class change-proneness. We show that, in almost all releases of Azureus and Eclipse, classes with

tipatterns is, however, out of scope of this study and will be treated in other works.

Premise. Code smells are conjectured in the literature to hinder object-oriented software evolution. Yet, despite the existence of many works on code smells and antipatterns, no previous work has contrasted the change-proneness of classes with code smells with this of other classes to study empirically the impact of code smells on this aspect of software evolution.

Goal. We want to investigate the relations between these code smells and three types of code evolution phenomena.

Study Context (1/2)

- Programs
  - 9 versions of Azureus (5,858,041 LOCs)
  - 13 versions of Eclipse (31,579,975 LOCs)
- Change history between each analysed releases in the programs' concurrent Versions System (CVS)

Study Context (2/2)

- 29 code smells

AbstractClass	ChildClass
ClassGlobalVariable	ClassOneMethod
ComplexClassOnly	ControllerClass
DataClass	FewMethods
FieldPrivate	FieldPublic
FunctionClass	HasChildren
LargeClass	LargeClassOnly
LongMethod	LongParameterListClass
LowCohesionOnly	ManyAttributes
MessageChainsClass	MethodNoParameter
MultipleInterface	NoInheritance
NoPolymorphism	NotAbstract
NotComplex	OneChildClass
ParentClassProvidesProtected	RareOverriding
TwoInheritance	

- Detected using our tool, DECOR

Study Design (1/3)

- RQ1: What is the relation between smells and change proneness?
  - H_{01}: the proportion of classes undergoing at least one change between two releases does not significantly differ between classes with code smells and other classes.
- RQ2: What is the relation between the number of smells in a class and its change-proneness?
  - H_{02}: the numbers of smells in change-prone classes are not significantly higher than the numbers of smells in classes that do not change.

Study Design (2/3)

- RQ3: What is the relation between particular kinds of smells and change proneness?
  - H_{03}: classes with particular kinds of code smells are not significantly more change-prone than other classes.

Results (1/6)

Results RQ1 (Azureus)

Releases	Smells-Changes	Smells-No Changes	No Smells-Changes	No Smells-No Changes	p-values	OR
3.1.0.0	220	1967	20	1433	$< \mathbf{0.01}$	8.01
3.1.1.0	564	1686	101	1381	$< \mathbf{0.01}$	4.57
4.0.0.0	83	2238	7	1519	$< \mathbf{0.01}$	8.05
4.0.0.2	106	2206	12	1510	$< \mathbf{0.01}$	6.04
4.0.0.4	435	1886	39	1484	$< \mathbf{0.01}$	8.77
4.1.0.0	50	2297	11	1533	$< \mathbf{0.01}$	3.03
4.1.0.2	112	2235	11	1533	$< \mathbf{0.01}$	6.98
4.1.0.4	112	2236	12	1532	$< \mathbf{0.01}$	6.39
4.2.0.0	37	2353	3	1580	$< \mathbf{0.01}$	8.28

Results (4/6)

- Results RQ2 (Eclipse)

Releases	M-W p	t-test p	Cohen d
1.0	0.79	**0.03**	0.06
2.0	$< \mathbf{0.01}$	$< \mathbf{0.01}$	−0.08
2.1.1	$< \mathbf{0.01}$	$< \mathbf{0.01}$	0.31
2.1.2	$< \mathbf{0.01}$	$< \mathbf{0.01}$	0.13
2.1.3	**0.04**	$< \mathbf{0.01}$	0.07
3.0	0.07	0.10	0.03
3.0.1	0.11	0.26	−0.03
3.0.2	0.12	0.28	−0.02
3.2	$< \mathbf{0.01}$	$< \mathbf{0.01}$	0.41
3.2.1	$< \mathbf{0.01}$	$< \mathbf{0.01}$	0.29
3.2.2	$< \mathbf{0.01}$	$< \mathbf{0.01}$	0.25
3.3	$< \mathbf{0.01}$	$< \mathbf{0.01}$	0.41
3.3.1	$< \mathbf{0.01}$	$< \mathbf{0.01}$	0.18

Results (6/6)

- Results RQ3 (Eclipse)

Smells	Proneness to Changes
AbstractClass	1
ChildClass	6
ClassGlobalVariable	2
ClassOneMethod	4
ComplexClassOnly	8
ControllerClass	4
DataClass	4
FewMethods	2
FieldPrivate	6
FieldPublic	8
FunctionClass	1
HasChildren	**11**
LargeClass	8
LargeClassOnly	–
LongMethod	9
LongParameterListClass	6
LowCohesionOnly	5
ManyAttributes	9
MessageChainsClass	**10**
MethodNoParameter	8
MultipleInterface	5
NoInheritance	–
NoPolymorphism	3
NotAbstract	1
NotComplex	**10**
OneChildClass	2
ParentClassProvidesProtected	–
RareOverriding	4
TwoInheritance	–

Discussion

- Classes with smells are more change-prone, some odds ratio 3 to 8 times bigger for these classes.
- HasChildren, MessageChains, NotComplex, and NotAbstract lead almost consistently to change-prone classes.
- Existing smells are generally removed from the system while some new are introduced in the context of new features addition.

INVESTIGATING FAULT-PRONENESS

(The EMSE Paper)

RELATION BETWEEN ANTIPATTERNS AND FAULT-PRONENESS

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

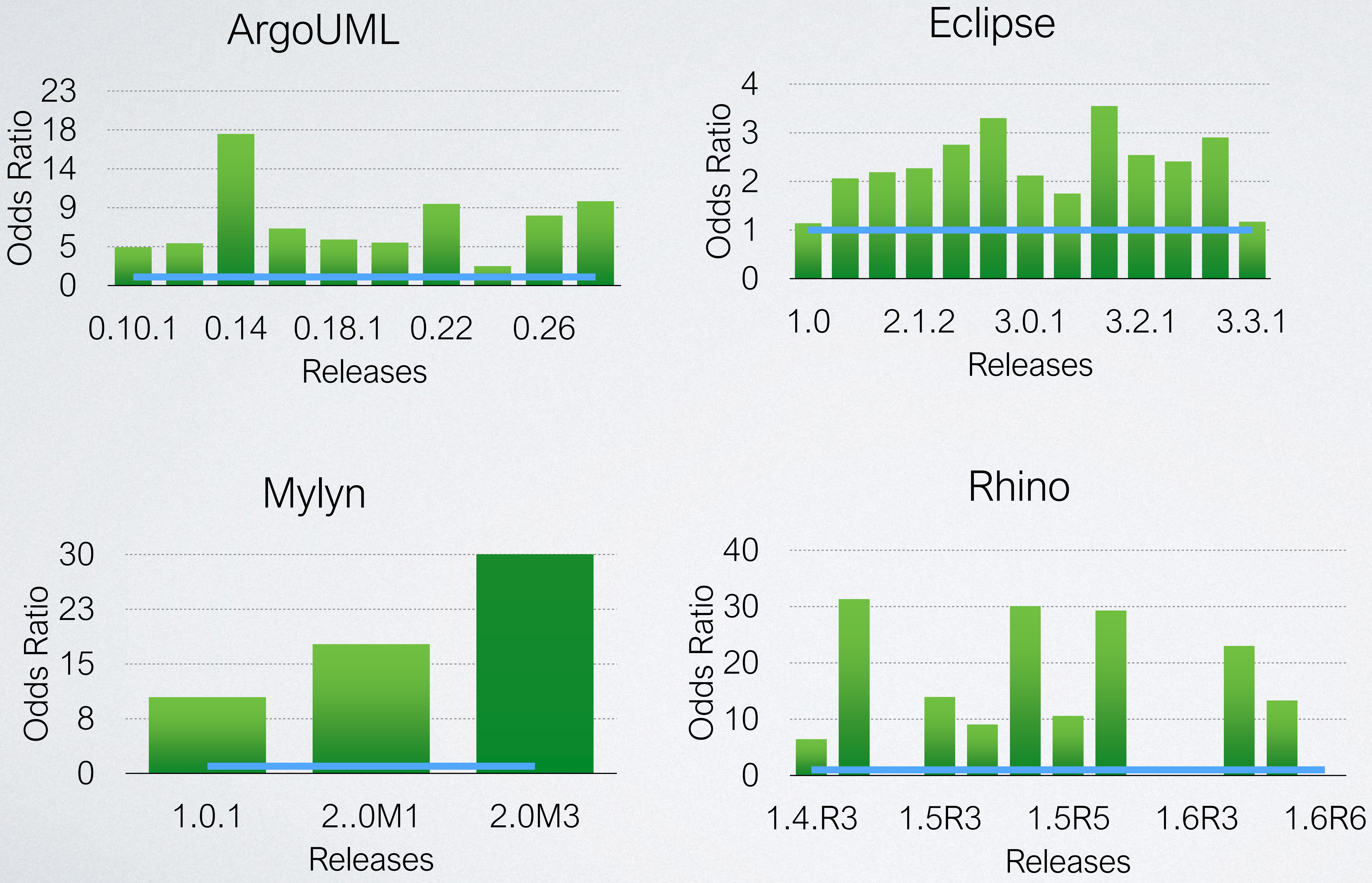
Foutse Khomh · Massimiliano Di Penta · Yann-Gaël Guéhéneuc · Giuliano Antoniol

Published online: 6 August 2011

Editor: Jim Whitehead

Abstract Antipatterns are poor design choices that are conjectured to make object-oriented systems harder to maintain. We investigate the impact of antipatterns on classes in object-oriented systems by studying the relation between the presence of antipatterns and the change- and fault-proneness of the classes. We detect 13 antipatterns in 54 releases of ArgoUML, Eclipse, Mylyn, and Rhino, and analyse (1)

ANTIPATTERNS AND FAULT-PRONENESS

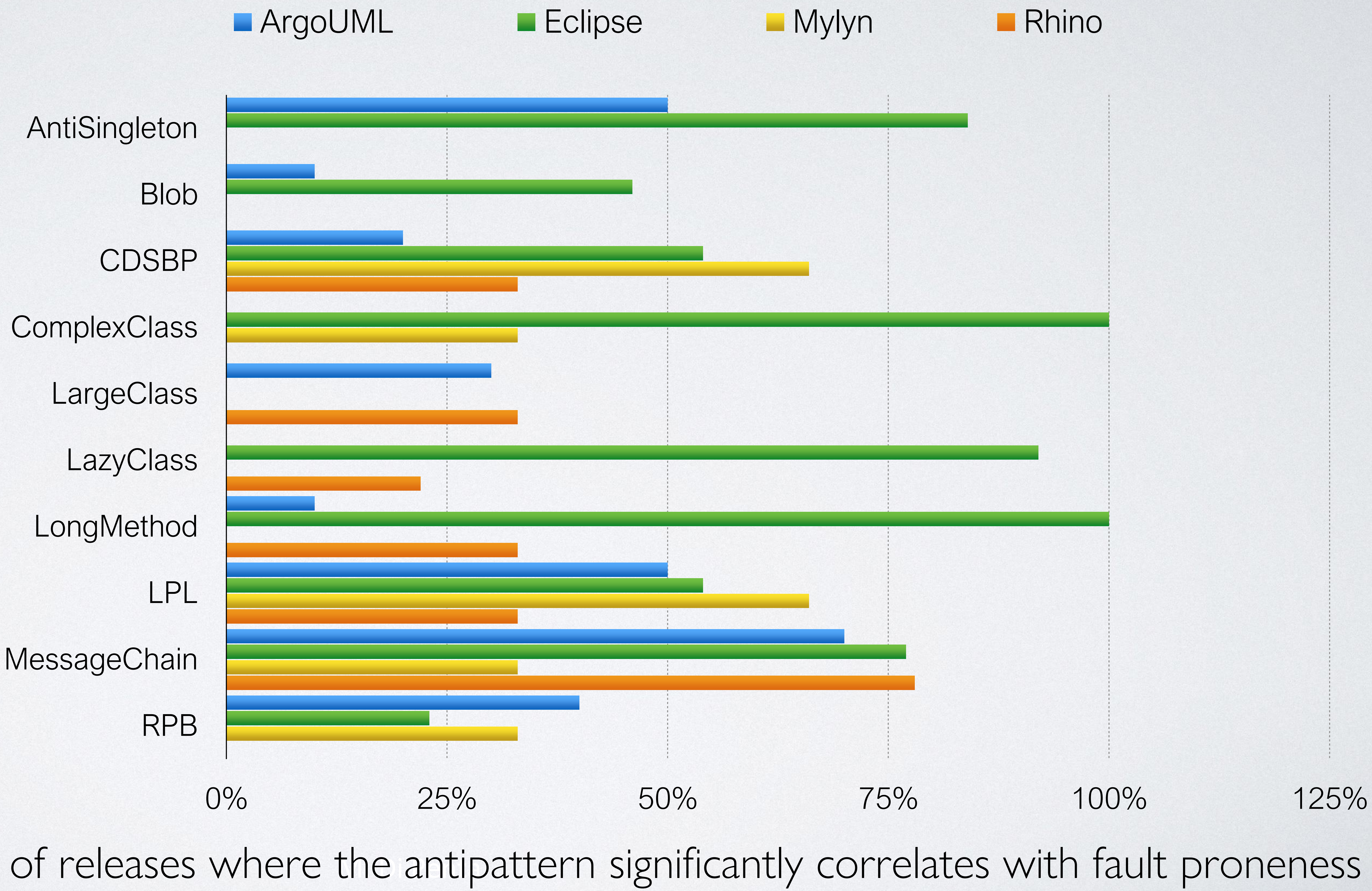


Antipattern classes have **up to 30 times more** chances to exhibit faults

Especially true for **coupling-related antipatterns** (e.g., Message Chains)

FAULT-PRONENESS: WHAT ANTIPATTERNS?

Especially true for coupling-related antipatterns (e.g., Message Chains)



INVESTIGATING ENERGY-EFFICIENCY

(The TSE Paper)

ANTIPATTERNS AND ENERGY-EFFICIENCY

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. X, NO. X, SEPTEMBER 2016 1

EARMO: An Energy-Aware Refactoring Approach for Mobile Apps

Rodrigo Morales, *Member, IEEE,* Rubén Saborido, *Member, IEEE,* Foutse Khomh, *Member, IEEE,* Francisco Chicano, and Giuliano Antoniol, *Senior Member, IEEE*

Abstract—The energy consumption of mobile apps is a trending topic and researchers are actively investigating the role of coding practices on energy consumption. Recent studies suggest that design choices can conflict with energy consumption. Therefore, it is important to take into account energy consumption when evolving the design of a mobile app. In this paper, we analyze the impact of eight type of anti-patterns on a testbed of 20 android apps extracted from F-Droid. We propose EARMO, a novel anti-pattern correction approach that accounts for energy consumption when refactoring mobile anti-patterns. We evaluate EARMO using three multiobjective search-based algorithms. The obtained results show that EARMO can generate refactoring recommendations in less than a minute, and remove a median of 84% of anti-patterns. Moreover, EARMO extended the battery life of a mobile phone by up to 29 minutes when running in isolation a refactored multimedia app with default settings (no WiFi, no location services, and minimum screen brightness). Finally, we conducted a qualitative study with developers of our studied apps, to assess the refactoring recommendations made by EARMO. Developers found 68% of refactorings suggested by EARMO to be very relevant.

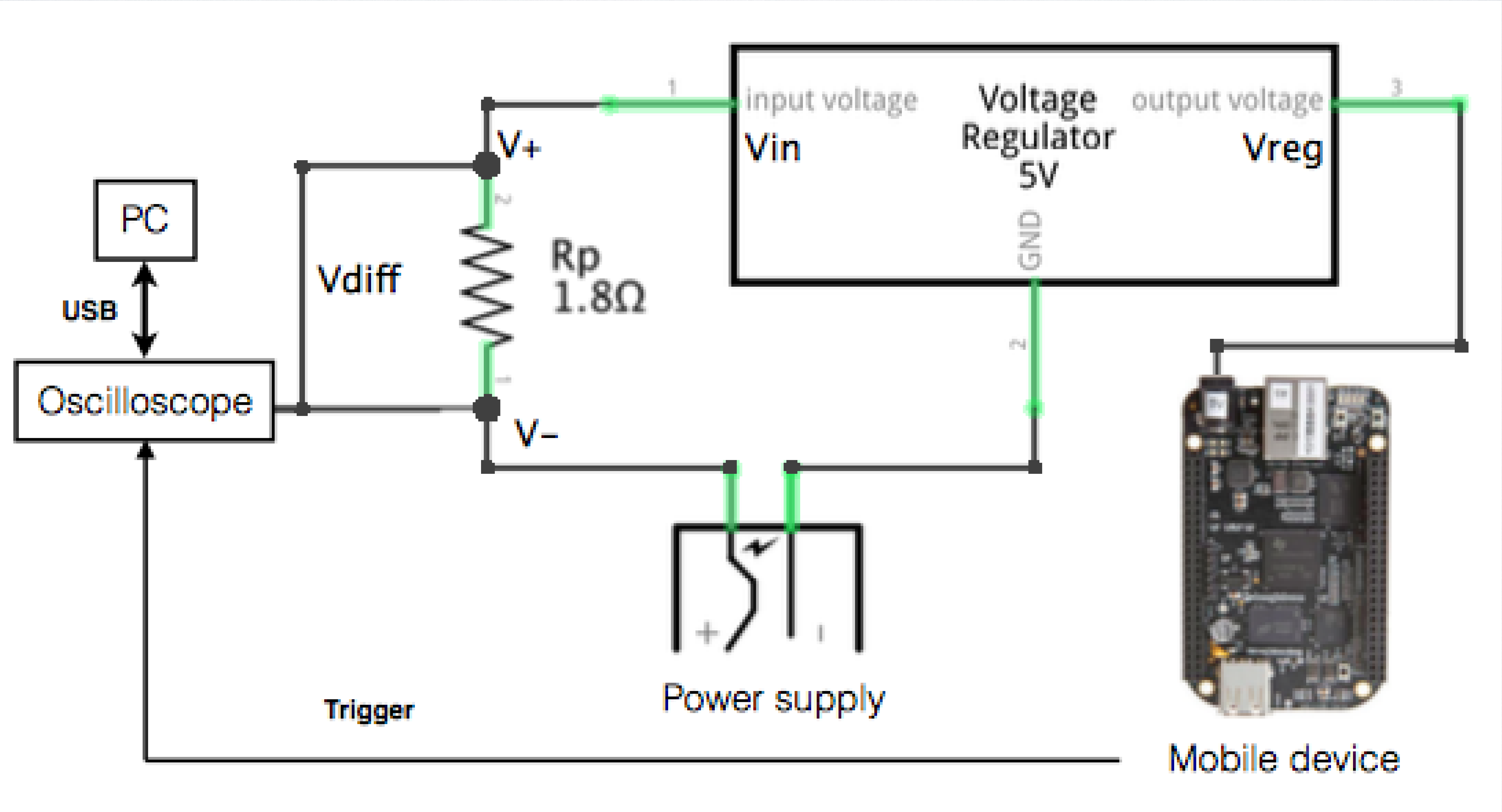
Index Terms—Software maintenance; Refactoring; Anti-patterns; Mobile apps; Energy consumption; Search-based Software Engineering

1 Introduction

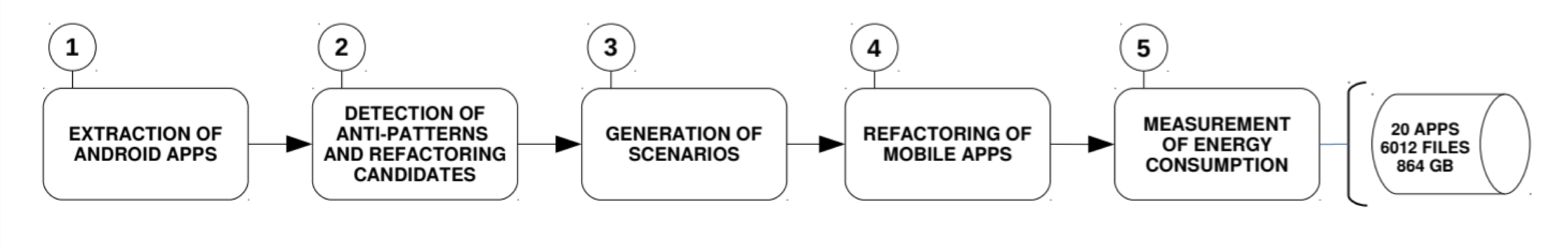
DURING the last five years, and with the exponential growth of the market of mobile apps [1], software

class, which is a large and complex class that centralizes most of the responsibilities of an app, while using the

ENERGY MEASUREMENT



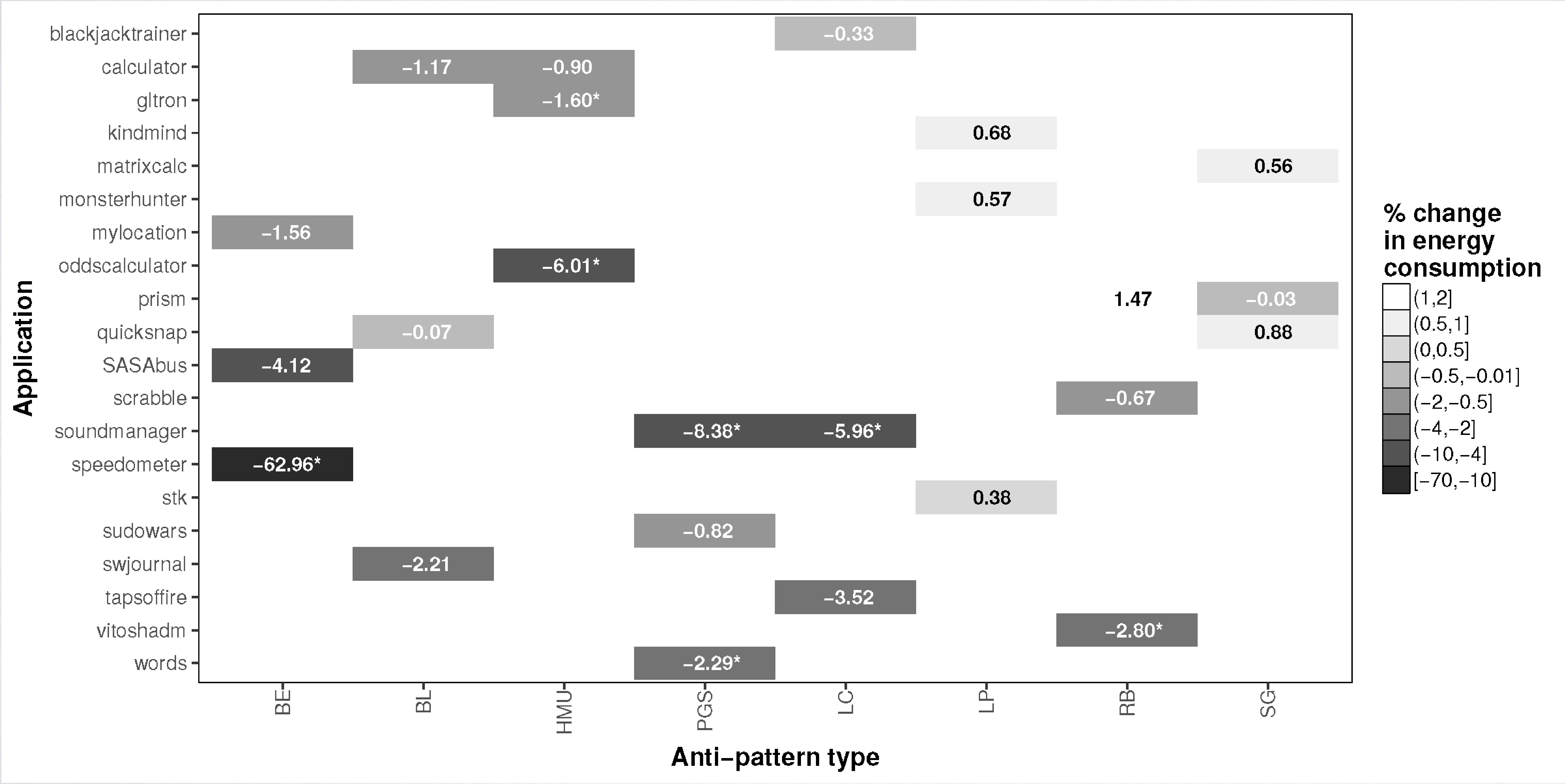
ENERGY MEASUREMENT



Blob (BL)
Lazy Class (LC)
Long-parameter list (LP)
Refused Bequest (RB)
Speculative Generality (SG)

Binding Resources too early (BE)
HashMap usage (HMU)
Private getters and setters (PGS)

ANTIPATTERNS AND ENERGY EFFICIENCY



Removing **Binding resources too early**, **Private getters and setters**, **Refused Bequest**, and **Lazy Class** anti-patterns can improve energy efficiency

DO THEY REALLY SMELL THAT BAD?

DESIGN FLAWS CORRELATE WITH SOFTWARE DEFECTS INTRODUCTION

2010 10th International Conference on Quality Software

On the Impact of Design Flaws on Software Defects

Marco D'Ambros, Alberto Bacchelli, Michele Lanza
REVEAL @ Faculty of Informatics - University of Lugano, Switzerland
{marco.dambros, alberto.bacchelli, michele.lanza}@usi.ch

***Abstract*—The presence of design flaws in a software system has a negative impact on the quality of the software, as they indicate violations of design practices and principles, which make a software system harder to understand, maintain, and evolve. Software defects are tangible effects of poor software quality.**

In this paper we study the relationship between software defects and a number of design flaws. We found that, while some design flaws are more frequent, none of them can be considered more harmful with respect to software defects. We also analyzed the correlation between the introduction of new flaws and the generation of defects.

***Index Terms*—Software quality and design; Software defects**

I. Introduction

In the light of the increased complexity of today's software systems, it is no wonder that maintenance and evolution claim 90% of the total software costs [1]. In this context, much effort has been devoted to find approaches capable of detecting parts of the source code that are likely to be harder to maintain, or to be more related to defects. Source code entities that have design flaws are good candidates, since these are known to have a negative impact on quality attributes such as flexibility or maintainability [2]. However, simple source code metrics are not capable of identifying poorly designed parts, because they must be analyzed and considered in the context in which

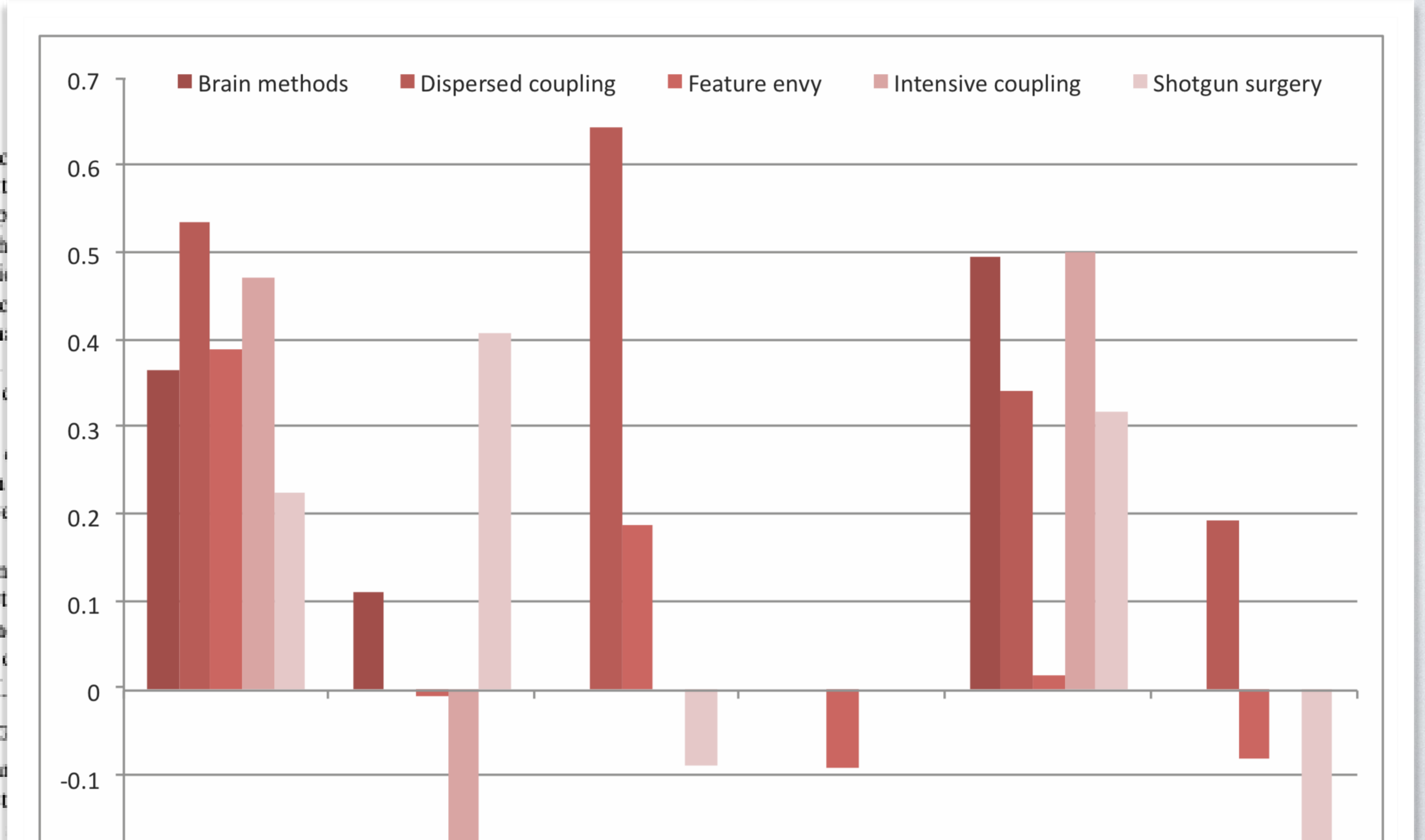
different. Our goal is to compare whether and ho
characteristics induct, influence, or alleviate different

Although our goal is clearly defined, we do not exp
able to fully determine it with six case studies, even th
consider the comprehensive data from all the histori
software systems analyzed. However, we do expect to
some useful evidence that can contribute to the an
the relationship between different design flaws, and
design flaws and software defects, and also, help
appropriate questions to ask in future case studies.

Structure of the paper In Section II we introduce
strategy: the technique we employ to identify design
software systems. In Section III we explain how we
link, and process the actual data from source code
repositories, that we later use in Section IV to con
core of the experiment. In Section V we outline the t
the validity of this study. In Section VI we analyze th
research on detecting design flaws and in the field
analysis and prediction. We conclude in Section VII.

II. Design Flaws and Detection Strateg

As opposed to object-oriented metrics [10], which a
measures of size (e.g., lines of code, number of met
complexity (e.g., McCabe cyclomatic complexity) of



SMELLS LIKE TEEN SPIRIT…

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

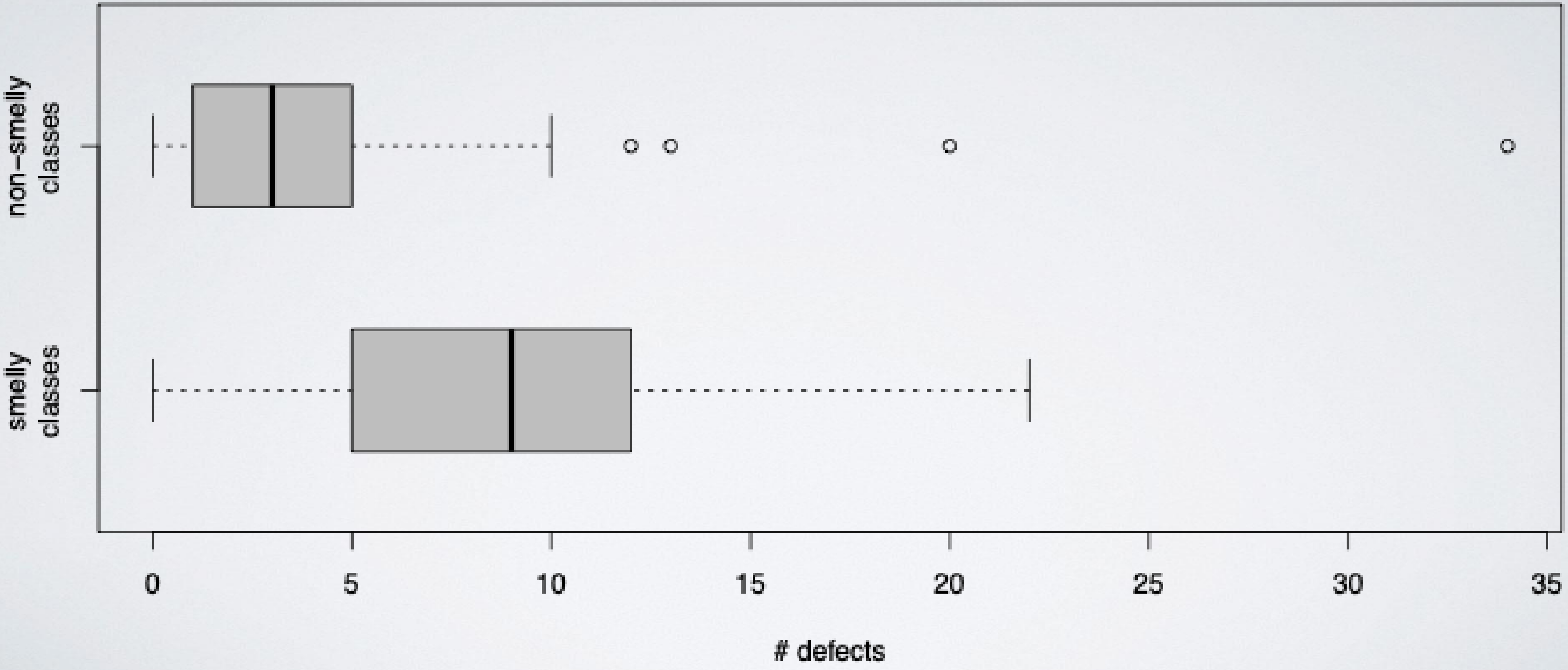
Fabio Palomba*, Marco Zanoni†, Francesca Arcelli Fontana†, Andrea De Lucia*, Rocco Oliveto‡
*University of Salerno, Italy, †University of Milano-Bicocca, Italy, ‡University of Molise, Italy
fpalomba@unisa.it, marco.zanoni@disco.unimib.it, arcelli@disco.unimib.it, adelucia@unisa.it, rocco.oliveto@unimol.it

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

prediction model can contribute to the correct classification of the buggyness of such a component. To verify this conjecture, we use the intensity index (*i.e.,* a metric able to estimate the severity of a code smell) defined by Arcelli Fontana *et al.* [31] to build a bug prediction model that takes into account the presence and the severity of design problems affecting a code component. Specifically, we evaluate the predictive power of the intensity index by adding it in a bug prediction model based on structural quality metrics [32], and comparing its accuracy against the one achieved by the baseline model on six large Java open source systems. We also quantified the gain provided by the addition of the intensity index with respect to the other structural metrics in the model, including the ones used to compute the intensity. Finally, we report further analyses aimed at understanding (i) the accuracy of a model

Smell intensity **more important** than other metrics for predicting fault-proneness

DEFECT-PRONENESS

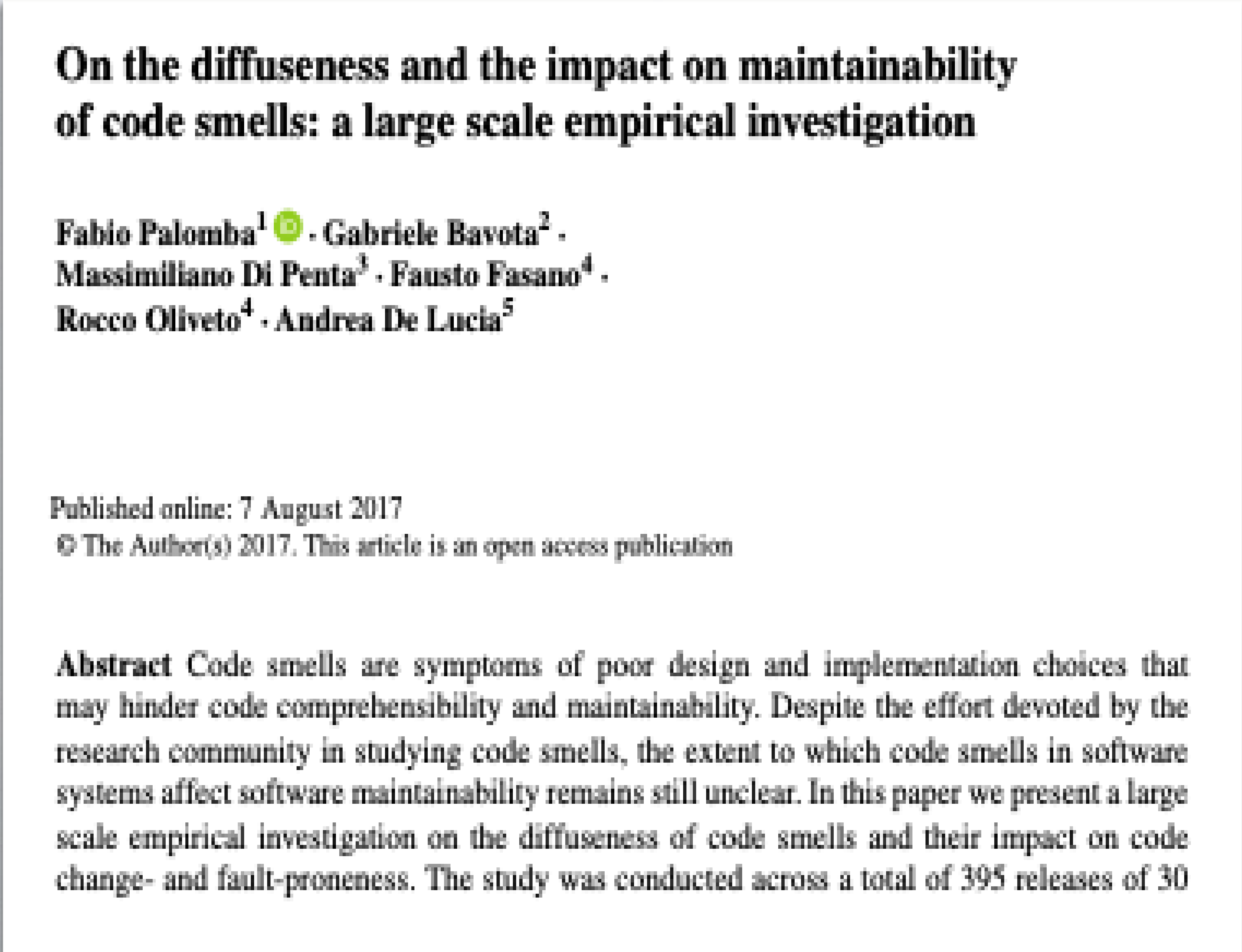


DO THEY REALLY SMELL THAT BAD?

YES, BUT…

CODE SMELL DIFFUSENESS

Coupling-related code smells are generally poorly diffused



On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation

Fabio Palomba[1] · Gabriele Bavota[2] · Massimiliano Di Penta[3] · Fausto Fasano[4] · Rocco Oliveto[4] · Andrea De Lucia[5]

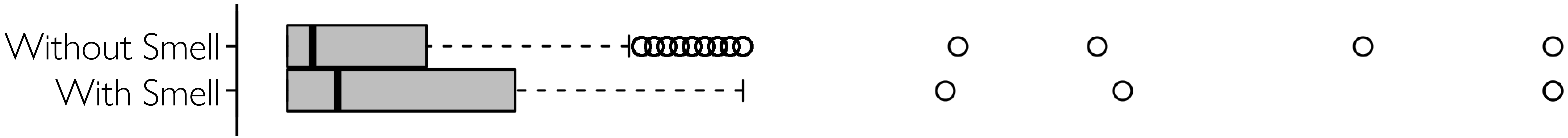
Published online: 7 August 2017
© The Author(s) 2017. This article is an open access publication

Abstract Code smells are symptoms of poor design and implementation choices that may hinder code comprehensibility and maintainability. Despite the effort devoted by the research community in studying code smells, the extent to which code smells in software systems affect software maintainability remains still unclear. In this paper we present a large scale empirical investigation on the diffuseness of code smells and their impact on code change- and fault-proneness. The study was conducted across a total of 395 releases of 30

13%

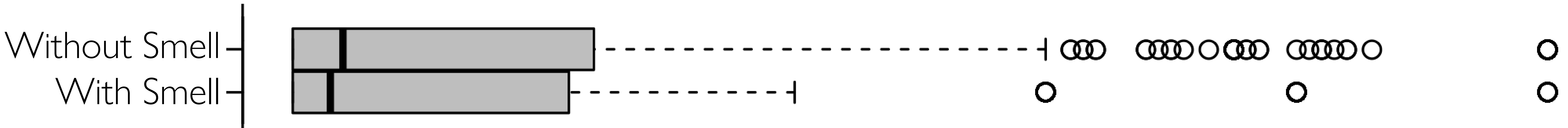
of the considered releases contained a **Message Chains** instance

LONGITUDINAL ANALYSIS



Message Chains

Long Parameter List



WHAT ABOUT THE MAGNITUDE OF THE EFFECT?

SMALL EFFECT, REFACTORING MIGHT BE WORTHLESS

Some Code Smells Have a Significant but Small Effect on Faults

TRACY HALL, Brunel University
MIN ZHANG, DAVID BOWES, and YI SUN, University of Hertfordshire

We investigate the relationship between faults and five of Fowler et al.'s least-studied smells in code: Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man. We developed a tool to detect these five smells in three open-source systems: Eclipse, ArgoUML, and Apache Commons. We collected fault data from the change and fault repositories of each system. We built Negative Binomial regression models to analyse the relationships between smells and faults and report the McFadden effect size of those relationships. Our results suggest that Switch Statements had no effect on faults in any of the three systems; Message Chains increased faults in two systems; Message Chains which occurred in larger files reduced faults; Data Clumps reduced faults in Apache and Eclipse but increased faults in ArgoUML; Middle Man reduced faults only in ArgoUML, and Speculative Generality reduced faults only in Eclipse. File size alone affects faults in some systems but not in all systems. Where smells did significantly affect faults, the size of that effect was small (always under 10 percent). Our findings suggest that some smells do indicate fault-prone code in some circumstances but that the effect that these smells have on faults is small. Our findings also show that smells have different effects on different systems. We conclude that arbitrary refactoring is unlikely to significantly reduce fault-proneness and in some cases may increase fault-proneness.

CODE SMELLS ARE DISCONNECTED FROM ARCHITECTURAL PROBLEMS

Are Automatically-Detected Code Anomalies Relevant to Architectural Modularity?

An Exploratory Analysis of Evolving Systems

Isela Macia[1], Joshua Garcia[2], Daniel Popescu[2], Alessandro Garcia[1], Nenad Medvidovic[2], Arndt von Staa[1]

[1]Opus Group, LES, Informatics Department, PUC-Rio, RJ, Brazil
[2]University of Southern California, Los Angeles, CA, USA

{ibertran, afgarcia, arndt}@inf.puc-rio.br, {joshuaga, dpopescu, neno}@usc.edu

ABSTRACT

As software systems are maintained, their architecture modularity often degrades through architectural erosion and drift. More directly, however, the modularity of software implementations degrades through the introduction of code anomalies, informally known as code smells. A number of strategies have been developed for supporting the automatic identification of implementation anomalies when only the source code is available. However, it is still unknown how reliable these strategies are when revealing code anomalies related to erosion and drift processes. In this paper, we present an exploratory analysis that investigates to what extent the automatically-detected code anomalies are related to problems that occur with an evolving system's architecture. We analyzed code anomaly occurrences in 38 versions of 5 applications using existing detection strategies. The outcome of our evaluation suggests that many of the code anomalies detected by the employed strategies were not related to architectural problems. Even worse, over 50% of the anomalies not observed by the employed techniques (false negatives) were found to be correlated with architectural problems.

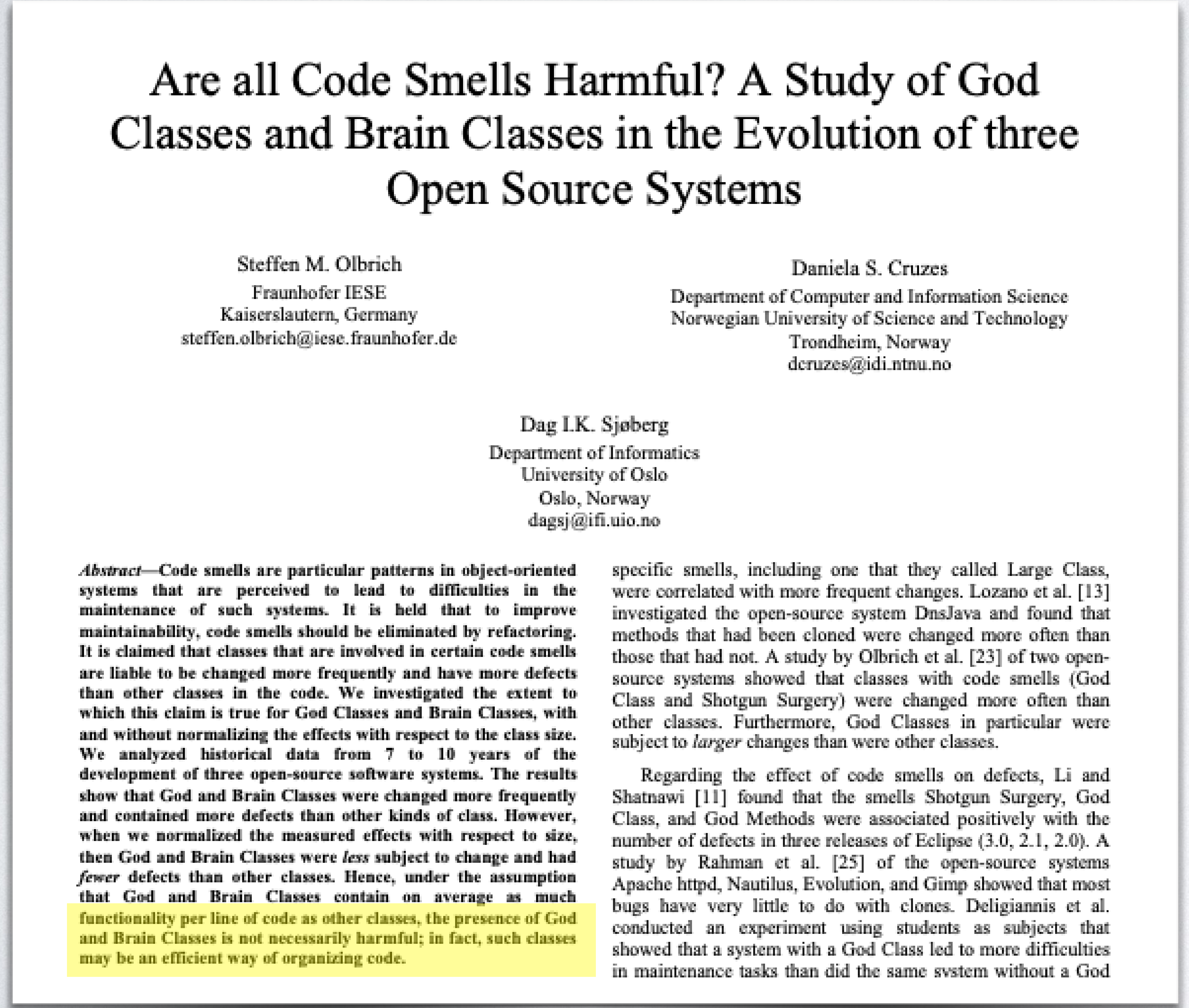
of modularization technique, including object-oriented programming [31] and aspect-oriented programming [19]. Code anomalies are often considered as key indicators of architectural degradation [13]. Hence, if these code anomalies are not systematically removed, the system's architectures may degrade due to erosion or drift [16]. Architectural erosion occurs when architectural violations are introduced, whereas drift is the realization of unintended design decisions also known as architectural anomalies [39].

The detection of architecturally-relevant code anomalies is particularly challenging when architectural designs are absent or obsolete, which is a common situation in evolving software projects. A complicating factor is that, due to time constraints, developers often need to concentrate on the most relevant anomalies. In other words, they should focus on code anomalies that are actually contributing to architecture erosion or drift. Let's consider a simple example of code anomaly, such as *God Class* [27]. Occurrences of *God Class* only cause harm to the architectural modularity when their realization of multiple concerns introduce undesirable dependencies between architecture elements (e.g., multiple architecture layers). Therefore, such *God Class* instances require closer, more immediate attention

Smells not reflected in architectural problems

Architectural problems not discovered by smell detectors

SMELLS ARE EFFICIENT WAYS TO ORGANIZE THE SOURCE CODE...



Are all Code Smells Harmful? A Study of God Classes and Brain Classes in the Evolution of three Open Source Systems

Steffen M. Olbrich
Fraunhofer IESE
Kaiserslautern, Germany
steffen.olbrich@iese.fraunhofer.de

Daniela S. Cruzes
Department of Computer and Information Science
Norwegian University of Science and Technology
Trondheim, Norway
dcruzes@idi.ntnu.no

Dag I.K. Sjøberg
Department of Informatics
University of Oslo
Oslo, Norway
dagsj@ifi.uio.no

***Abstract*—Code smells are particular patterns in object-oriented systems that are perceived to lead to difficulties in the maintenance of such systems. It is held that to improve maintainability, code smells should be eliminated by refactoring. It is claimed that classes that are involved in certain code smells are liable to be changed more frequently and have more defects than other classes in the code. We investigated the extent to which this claim is true for God Classes and Brain Classes, with and without normalizing the effects with respect to the class size. We analyzed historical data from 7 to 10 years of the development of three open-source software systems. The results show that God and Brain Classes were changed more frequently and contained more defects than other kinds of class. However, when we normalized the measured effects with respect to size, then God and Brain Classes were *less* subject to change and had *fewer* defects than other classes. Hence, under the assumption that God and Brain Classes contain on average as much functionality per line of code as other classes, the presence of God and Brain Classes is not necessarily harmful; in fact, such classes may be an efficient way of organizing code.**

specific smells, including one that they called Large Class, were correlated with more frequent changes. Lozano et al. [13] investigated the open-source system DnsJava and found that methods that had been cloned were changed more often than those that had not. A study by Olbrich et al. [23] of two open-source systems showed that classes with code smells (God Class and Shotgun Surgery) were changed more often than other classes. Furthermore, God Classes in particular were subject to *larger* changes than were other classes.

Regarding the effect of code smells on defects, Li and Shatnawi [11] found that the smells Shotgun Surgery, God Class, and God Methods were associated positively with the number of defects in three releases of Eclipse (3.0, 2.1, 2.0). A study by Rahman et al. [25] of the open-source systems Apache httpd, Nautilus, Evolution, and Gimp showed that most bugs have very little to do with clones. Deligiannis et al. conducted an experiment using students as subjects that showed that a system with a God Class led to more difficulties in maintenance tasks than did the same svstem without a God

Junior developers perform better when working with source code with a centralized control style

SMELLS DO NOT INCREASE MAINTENANCE EFFORT

Quantifying the Effect of Code Smells on Maintenance Effort

Dag I.K. Sjøberg, *Member, IEEE*, Aiko Yamashita, *Student Member, IEEE*, Bente C.D. Anda, Audris Mockus, *Member, IEEE*, and Tore Dybå, *Member, IEEE*

Abstract—Context: Code smells are assumed to indicate bad design that leads to less maintainable code. However, this assumption has not been investigated in controlled studies with professional software developers. **Aim:** This paper investigates the relationship between code smells and maintenance effort. **Method:** Six developers were hired to perform three maintenance tasks each on four functionally equivalent Java systems originally implemented by different companies. Each developer spent three to four weeks. In total, they modified 298 Java files in the four systems. An Eclipse IDE plug-in measured the exact amount of time a developer spent maintaining each file. Regression analysis was used to explain the effort using file properties, including the number of smells. **Result:** None of the 12 investigated smells was significantly associated with increased effort after we adjusted for file size and the number of changes; Refused Bequest was significantly associated with **decreased** effort. File size and the number of changes explained almost all of the modeled variation in effort. **Conclusion:** The effects of the 12 smells on maintenance effort were limited. To reduce maintenance effort, a focus on reducing code size and the work practices that limit the number of changes may be more beneficial than refactoring code smells.

ON THE LIFE AND DEATH OF CODE SMELLS

Tracking Design Smells: Lessons from a Study of God Classes

Stéphane Vaucher   Foutse Khomh   Naouel Moha   Yann-Gaël Guéhéneuc

GEODES / Ptidej Team   *Triskell Team*   *Ptidej Team*

D

Un

Email: {vauc

***Abstract*—"G**
type of large cla
Often a God cla
are incremental
of its evolution.
of bad code tha
software quality
design as the b
for example, the

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

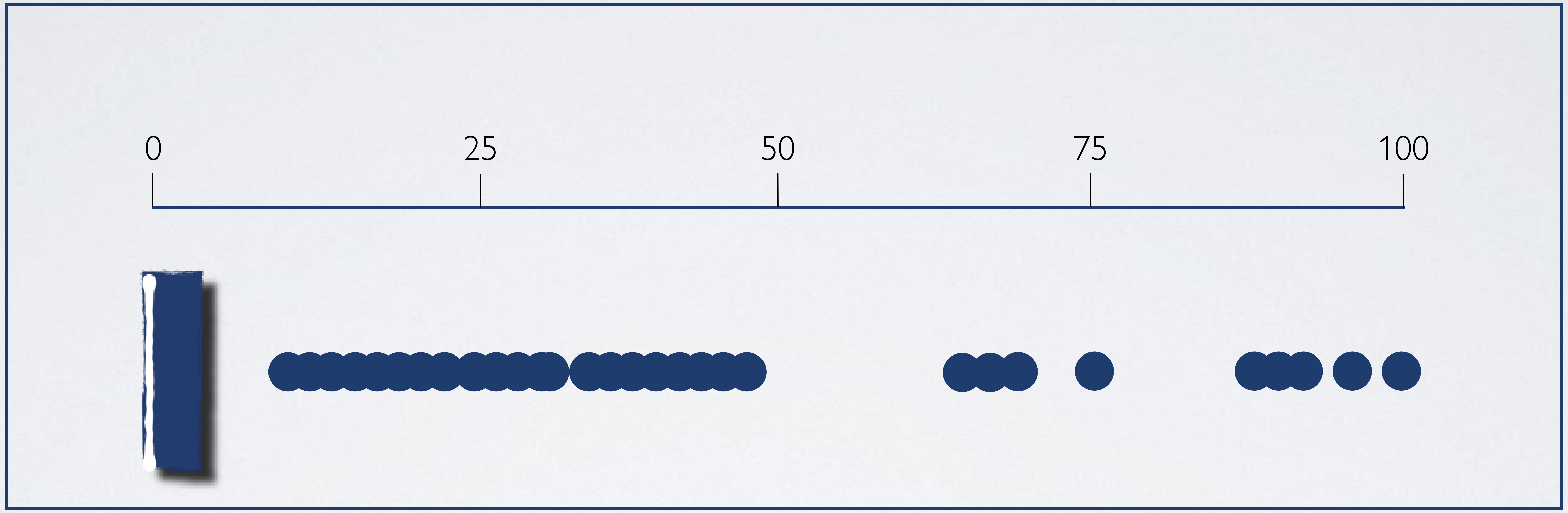
Michele Tufano[1], Fabio Palomba[2], Gabriele Bavota[3]
Rocco Oliveto[4], Massimiliano Di Penta[5], Andrea De Lucia[2], Denys Poshyvanyk[1]

[1]The College of William and Mary, Williamsburg, VA, USA [2]University of Salerno, Fisciano (SA), Italy,
[3]Università della Svizzera italiana (USI), Switzerland, [4]University of Molise, Pesche (IS), Italy,
[5]University of Sannio, Benevento (BN), Italy

mtufano@email.wm.edu, fpalomba@unisa.it, gabriele.bavota@usi.ch
rocco.oliveto@unimol.it, dipenta@unisannio.it, adelucia@unisa.it, denys@cs.wm.edu

WHEN BLOBS ARE INTRODUCED

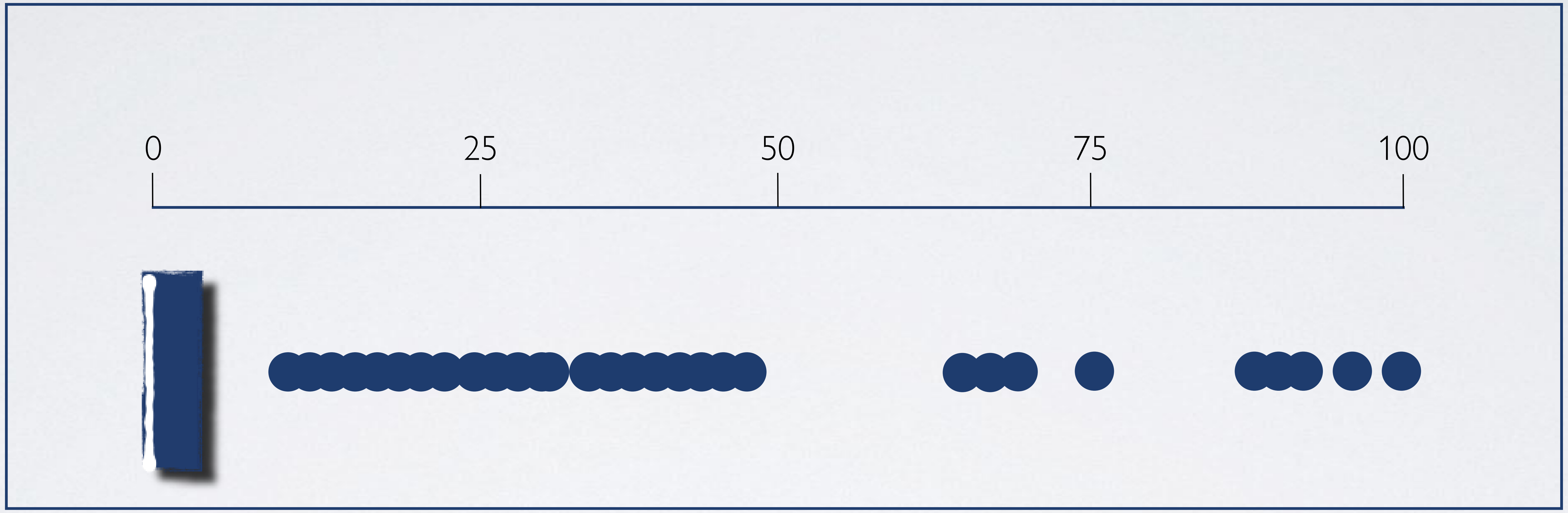
Commits required to a class for becoming smell



Generally, blobs affect a class since its creation

WHEN BLOBS ARE INTRODUCED

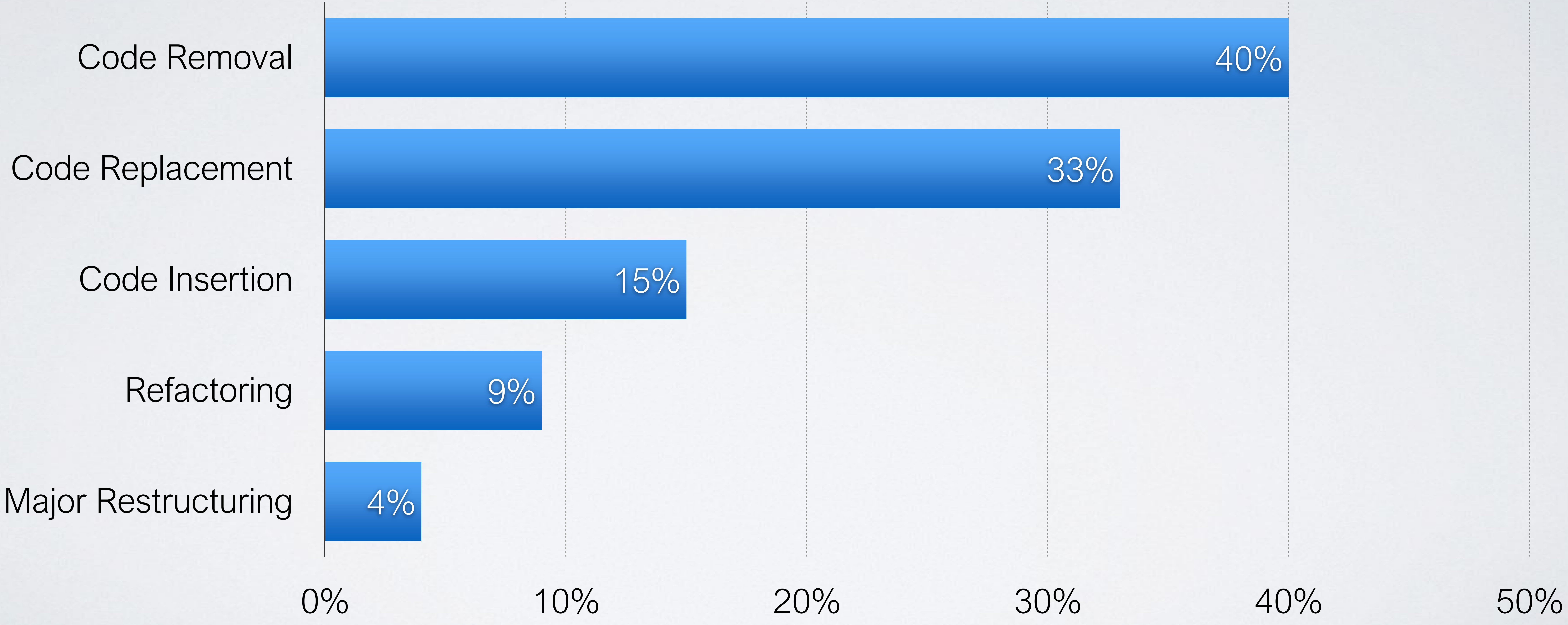
Commits required to a class for becoming smell



Generally, blobs affect a class since its creation

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

SMELL REMOVAL



CONFIRMED BY A RECENT STUDY THROUGH INTERVIEWS

"Refactoring activity is mainly driven by changes in the requirements and much less by code smell resolution."

Why We Refactor? Confessions of GitHub Contributors

Danilo Silva
Universidade Federal de Minas Gerais, Brazil
danilofs@dcc.ufmg.br

Nikolaos Tsantalis
Concordia University
Montreal, Canada
tsantalis@cse.concordia.ca

Marco Tulio Valente
Universidade Federal de Minas Gerais, Brazil
mtov@dcc.ufmg.br

ABSTRACT

Refactoring is a widespread practice that helps developers to improve the maintainability and readability of their code. However, there is a limited number of studies empirically investigating the actual motivations behind specific refactoring operations applied by developers. To fill this gap,

As a second example, MOVE METHOD is associated to smells like Feature Envy and Shotgun Surgery [10].

There is a limited number of studies investigating the real motivations driving the refactoring practice based on interviews and feedback from actual developers. Kim et al. [17] explicitly asked developers "in which situations do you perform refactorings?" and recorded 10 code symptoms that

…SO DEVELOPERS FOCUS ON OTHER THINGS WHILE DOING REFACTORING…

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.**

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?

BEYOND CODE SMELLS…

TEST SMELLS

An Empirical Analysis of the Distribution of Unit Test Smells and Their Impact on Software Maintenance

Gabriele Bavota[1], Abdallah Qusef[1], Rocco Oliveto[2], Andrea De Lucia[1], David Binkley[3]

An Empirical Investigation into the Nature of Test Smells

Michele Tufano[1], Fabio Palomba[2], Gabriele Bavota[3], Massimiliano Di Penta[4], Rocco Oliveto[5], Andrea De Lucia[2], Denys Poshyvanyk[1]

Affect maintainability…

…but developers rarely perceive them as important

LINGUISTIC ANTIPATTERNS

Linguistic antipatterns: what they are and how developers perceive them

Venera Arnaoudova · Massimiliano Di Penta · Giuliano Antoniol

Published online: 29 January 2015
© Springer Science+Business Media New York 2015

Abstract Antipatterns are known as poor solutions to recurring problems. For example, Brown et al. and Fowler define practices concerning poor design or implementation solutions. However, we know that the source code lexicon is part of the factors that affect the psychological complexity of a program, i.e., factors that make a program difficult to understand and maintain by humans. The aim of this work is to identify recurring poor practices related to inconsistencies among the naming, documentation, and implementation of an entity—called Linguistic Antipatterns (LAs)—that may impair program understanding.

Perceived as serious concerns by developers

When present in APIs, correlate with the increase of Stack Overflow questions [Aghajani et al.]

SQL SMELLS

Tool for detecting query antipatterns from Bill Karwins SQL Antipatterns catalog

A Static Code Smell Detector for SQL Queries Embedded in Java Code

Csaba Nagy*, Anthony Cleve†
PReCISE Research Center, University of Namur, Belgium

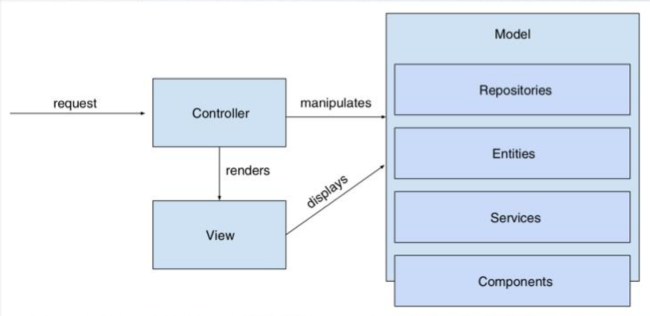
CODE SMELLS IN MODEL-VIEW CONTROLLER ARCHITECURES

Code smells for Model-View-Controller architectures

Maurício Aniche[1] · Gabriele Bavota[2] · Christoph Treude[3] · Marco Aurélio Gerosa[4] · Arie van Deursen[1]

Published online: 12 September 2017
© The Author(s) 2017. This article is an open access publication

Abstract Previous studies have shown the negative effects that low-quality code can have on maintainability proxies, such as code change- and defect-proneness. One of the symptoms of low-quality code are code smells, defined as sub-optimal implementation choices. While this definition is quite general and seems to suggest a wide spectrum of smells that can affect software systems, the research literature mostly focuses on the set of smells defined in



SERVICE ORIENTED ARCHITECTURE SMELLS

Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces

Mathieu Nayrolles, Naouel Moha, and Petko Valtchev

Investigating the Change-proneness of Service Patterns and Antipatterns

Francis Palma, Le An, Foutse Khomh, Naouel Moha and Yann-Gaël Guéhéneuc

ENERGY SMELLS

470 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 44, NO. 5, MAY 2018

EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps

Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury

Abstract—Increased usage of mobile devices, such as smartphones and tablets, has led to widespread popularity and usage of mobile apps. If not carefully developed, such apps may demonstrate energy-inefficient behaviour, where one or more energy-intensive hardware components (such as Wifi, GPS, etc) are left in a high-power state, even when no apps are using these components. We refer to such kind of energy-inefficiencies as energy bugs. Executing an app with an energy bug causes the mobile device to exhibit poor energy consumption behaviour and a drastically shortened battery life. Since mobile apps can have huge input domains, therefore exhaustive exploration is often impractical. We believe that there is a need for a framework that can systematically detect and fix energy bugs in mobile apps in a scalable fashion. To address this need, we have developed EnergyPatch, a framework that uses a combination of static and dynamic analysis techniques to detect, validate and repair energy bugs in Android apps. The use of a light-weight, static analysis technique enables EnergyPatch to quickly narrow down to the potential program paths along which energy bugs may occur. Subsequent exploration of these potentially buggy program paths using a dynamic analysis technique helps in validations of the reported bugs and to generate test cases. Finally, EnergyPatch generates repair expressions to fix the validated energy bugs. Evaluation with real-life apps from repositories such as F-droid and Github, shows that EnergyPatch is scalable and can produce results in reasonable amount of time. Additionally, we observed that the repair expressions generated by EnergyPatch could bring down the energy consumption on tested apps up to 60 percent.

INFRASTRUCTURE-AS-CODE SMELLS

Does Your Configuration Code Smell?

Tushar Sharma, Marios Fragkoulis and Diomidis Spinellis
Dept of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
{tushar,mfg,dds}@aueb.gr

ABSTRACT
Infrastructure as Code (IaC) is the practice of specifying computing system configurations through code, and managing them through traditional software engineering methods. The wide adoption of configuration management and increasing size and complexity of the associated code, prompt for assessing, maintaining, and improving the configuration code's quality. In this context, traditional software engineering knowledge and best practices associated with code quality management can be leveraged to assess and manage

1. INTRODUCTION
Infrastructure as Code (IaC) [13] is the practice of specifying computing system configurations through code, automating system deployment, and managing the system configurations through traditional software engineering methods. For example, a server farm that contains numerous nodes with different hardware configurations and different software package requirements can be specified using configuration management languages such as Puppet [39], Chef [37], CFEngine [4], or Ansible [1] and deployed automatically

SMELLS IN CONTINUOUS INTEGRATION PIPELINES

Use and Misuse of Continuous Integration Features
An Empirical Study of Projects that (mis)use Travis CI

Keheliya Gallaba, *Student Member, IEEE,* and Shane McIntosh, *Member, IEEE*

Automated Reporting of Anti-Patterns and Decay in Continuous Integration

Carmine Vassallo, Department of Informatics, University of Zurich, Zurich, Switzerland, vassallo@ifi.uzh.ch
Sebastian Proksch, Department of Informatics, University of Zurich, Zurich, Switzerland, proksch@ifi.uzh.ch
Harald C. Gall, Department of Informatics, University of Zurich, Zurich, Switzerland, gall@ifi.uzh.ch
Massimiliano Di Penta, Department of Engineering, University of Sannio, Benevento, Italy, dipenta@unisannio.it

COMMUNITY SMELLS

Beyond Technical Aspects: How Do Community Smells Influence the Intensity of Code Smells?

Fabio Palomba, *Member, IEEE,* Damian A. Tamburri, *Member, IEEE,* Francesca Arcelli Fontana, *Member, IEEE,* Rocco Oliveto, *Member, IEEE,* Andy Zaidman, *Member, IEEE,* Alexander Serebrenik, *Senior Member, IEEE.*

Abstract—Code smells are poor implementation choices applied by developers during software evolution that often lead to critical flaws or failure. Much in the same way, community smells reflect the presence of organizational and socio-technical issues within a software community that may lead to additional project costs. Recent empirical studies provide evidence that community smells are often—if not always—connected to circumstances such as code smells. In this paper we look deeper into this connection by conducting a mixed-methods empirical study of 117 releases from 9 open-source systems. The qualitative and quantitative sides of our mixed-methods study were run in parallel and assume a mutually-confirmative connotation. On the one hand, we survey 162 developers of the 9 considered systems to investigate whether developers perceive relationship between community smells and the code smells found in those projects. On the other hand, we perform a fine-grained analysis into the 117 releases of our dataset to measure the extent to which community smells impact code smell intensity (i.e., criticality). We then propose a code smell intensity prediction model that relies on both technical and community-related aspects. The results of both sides of our mixed-methods study lead to one conclusion: community-related factors contribute to the intensity of code smells. This conclusion supports the joint use of community and code smells detection as a mechanism for the joint management of technical and social problems around software development communities.

TEST SMELLS

An Empirical Analysis of the Distribution of Unit Test Smells and Their Impact on Software Maintenance

Gabriele Bavota[1], Abdallah Qusef[1], Rocco Oliveto[2], Andrea De Lucia[1], David Binkley[3]
[1]University of Salerno, Fisciano (SA), Italy
[2]University of Molise, Pesche (IS), Italy
[3]Loyola University Maryland, Baltimore, USA
gbavota@unisa.it, aqusef@unisa.it, rocco.oliveto@unimol.it, adelucia@unisa.it, binkley@...

An Empirical Investigation into the Nature of Test Smells

Michele Tufano[1], Fabio Palomba[2], Gabriele Bavota[3], Massimiliano Di Penta[4]
Rocco Oliveto[5], Andrea De Lucia[2], Denys Poshyvanyk[1]
[1] The College of William and Mary, USA — [2] University of Salerno, Italy — [3] Università della Svizzera italiana (USI), Switzerland — [4] University of Sannio, Italy — [5] University of Molise, Italy

ABSTRACT

Affect maintainability…

…but developers rarely perceive them as important

LINGUISTIC ANTIPATTERNS

Linguistic antipatterns: what they are and how developers perceive them

Venera Arnaoudova · Massimiliano Di Penta · Giuliano Antoniol

Published online: 29 January 2015

Abstract Antipatterns are known as poor solutions to recurring problems. For example, Brown et al. and Fowler define practices concerning poor design or implementation solutions. However, we know that the source code lexicon is part of the factors that affect the psychological complexity of a program, i.e., factors that make a program difficult to understand and maintain by humans. The aim of this work is to identify recurring poor practices related to inconsistencies among the naming, documentation, and implementation of an entity—called Linguistic Antipatterns (LAs)—that may impair program understanding. To

Perceived as serious concerns by developers

When present in APIs, correlate with the increase of Stack Overflow questions [Aghajani et al.]

SQL SMELLS

Tool for detecting query antipatterns from Bill Karwins SQL Antipatterns catalog

A Static Code Smell Detector for SQL Queries Embedded in Java Code

Csaba Nagy*, Anthony Cleve†
PReCISE Research Center, University of Namur, Belgium
*csaba.nagy@unamur.be, †anthony.cleve@unamur.be

Abstract—A database plays a central role in the architecture of an information system, and the way it stores the data delimits its main features. However, it is not just the data that matters. The way it is handled, i.e., how the application communicates with the database is of critical importance too. Therefore the implementation of such a communication layer has to be reliable and efficient. SQL is a popular language to query a database, and modern technologies rely on it (or its dialects) as query strings embedded in the application code. In many languages (e.g. in Java), an embedded query is typically constructed through several string operations that obstruct developers in

technologies to communicate with a DB, while JDBC occurs as the only database framework in 56.3% of the projects [3].

Database access technologies intend to help developers in various ways. They make it easier to integrate the communication with the database into the application code, e.g., by providing a link between Java classes and database entities (e.g. ORMs), or merely by supporting to reuse and construct queries (e.g. prepared statements). However, as a drawback, a developer hardly sees the final SQL query that is, in the

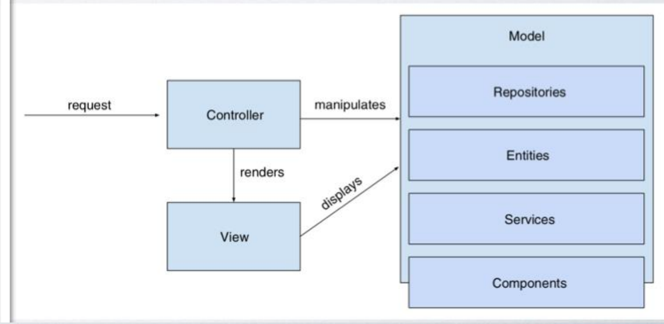
CODE SMELLS IN MODEL-VIEW CONTROLLER ARCHITECURES

Code smells for Model-View-Controller architectures

Maurício Aniche[1] · Gabriele Bavota[2] · Christoph Treude[3] · Marco Aurélio Gerosa[4] · Arie van Deursen[1]

Published online: 12 September 2017

Abstract Previous studies have shown the negative effects that low-quality code can have on maintainability proxies, such as code change- and defect-proneness. One of the symptoms of low-quality code are code smells, defined as sub-optimal implementation choices. While this definition is quite general and seems to suggest a wide spectrum of smells that can affect software systems, the research literature mostly focuses on the set of smells defined in



SERVICE ORIENTED ARCHITECTURE SMELLS

Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces

Mathieu Nayrolles, Naouel Moha, and Petko Valtchev

Investigating the Change-proneness of Service Patterns and Antipatterns

Francis Palma*†, Le An‡, Foutse Khomh‡, Naouel Moha† and Yann-Gaël Guéhéneuc*
*Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
†Latece, Département d'informatique, Université du Québec à Montréal, Canada
‡SWAT, DGIGL, École Polytechnique de Montréal, Canada
Email: {francis.palma, le.an, foutse.khomh, yann-gael.gueheneuc}@polymtl.ca, moha.naouel@uqam.ca

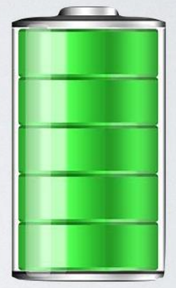
ENERGY SMELLS

470 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 44, NO. 5, MAY 2018

EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps

Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury

Abstract—Increased usage of mobile devices, such as smartphones and tablets, has led to widespread popularity and usage of mobile apps. If not carefully developed, such apps may demonstrate energy-inefficient behaviour, where one or more energy-intensive hardware components (such as Wifi, GPS, etc) are left in a high-power state, even when no apps are using these components. We refer to such kind of energy-inefficiencies as energy bugs. Executing an app with an energy bug causes the mobile device to exhibit poor energy consumption behaviour and a drastically shortened battery life. Since mobile apps can have huge input domains, therefore exhaustive exploration is often impractical. We believe that there is a need for a framework that can systematically detect and fix energy bugs in mobile apps in a scalable fashion. To address this need, we have developed EnergyPatch, a framework that uses a combination of static and dynamic analysis techniques to detect, validate and repair energy bugs in Android apps. The use of a light-weight, static analysis technique enables EnergyPatch to quickly narrow down to the potential program paths along which energy bugs may occur. Subsequent exploration of these potentially buggy program paths using a dynamic analysis technique helps in validations of the reported bugs and to generate test cases. Finally, EnergyPatch generates repair expressions to fix the validated energy bugs. Evaluation with real-life apps from repositories such as F-droid and Github, shows that EnergyPatch is scalable and can produce results in reasonable amount of time. Additionally, we observed that the repair expressions generated by EnergyPatch could bring down the energy consumption on tested apps up to 60 percent.



INFRASTRUCTURE-AS-CODE SMELLS

Does Your Configuration Code Smell?

Tushar Sharma, Marios Fragkoulis and Diomidis Spinellis
Dept of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
{tushar,mfg,dds}@aueb.gr

ABSTRACT

Infrastructure as Code (IaC) is the practice of specifying computing system configurations through code, and managing them through traditional software engineering methods. The wide adoption of configuration management and increasing size and complexity of the associated code, prompt for assessing, maintaining, and improving the configuration code's quality. In this context, traditional software engineering knowledge and best practices associated with code quality management can be leveraged to assess and manage

1. INTRODUCTION

Infrastructure as Code (IaC) [13] is the practice of specifying computing system configurations through code, automating system deployment, and managing the system configurations through traditional software engineering methods. For example, a server farm that contains numerous nodes with different hardware configurations and different software package requirements can be specified using configuration management languages such as Puppet [39], Chef [37], CFEngine [4], or Ansible [1] and deployed automatically

SMELLS IN CONTINUOUS INTEGRATION PIPELINES

Use and Misuse of Continuous Integration Features

An Empirical Study of Projects that (mis)use Travis CI

Keheliya Gallaba, *Student Member, IEEE*, and Shane McIntosh, *Member, IEEE*

Automated Reporting of Anti-Patterns and Decay in Continuous Integration

Carmine Vassallo, *Department of Informatics, University of Zurich*, Zurich, Switzerland, vassallo@ifi.uzh.ch
Sebastian Proksch, *Department of Informatics, University of Zurich*, Zurich, Switzerland, proksch@ifi.uzh.ch
Harald C. Gall, *Department of Informatics, University of Zurich*, Zurich, Switzerland, gall@ifi.uzh.ch
Massimiliano Di Penta, *Department of Engineering, University of Sannio*, Benevento, Italy, dipenta@unisannio.it

COMMUNITY SMELLS

Beyond Technical Aspects: How Do Community Smells Influence the Intensity of Code Smells?

Fabio Palomba, *Member, IEEE*, Damian A. Tamburri, *Member, IEEE*, Francesca Arcelli Fontana, *Member, IEEE*, Rocco Oliveto, *Member, IEEE*, Andy Zaidman, *Member, IEEE*, Alexander Serebrenik, *Senior Member, IEEE*.

Abstract—Code smells are poor implementation choices applied by developers during software evolution that often lead to critical flaws or failure. Much in the same way, community smells reflect the presence of organizational and socio-technical issues within a software community that may lead to additional project costs. Recent empirical studies provide evidence that community smells are often—if not always—connected to circumstances such as code smells. In this paper we look deeper into this connection by conducting a mixed-methods empirical study of 117 releases from 9 open-source systems. The qualitative and quantitative sides of our mixed-methods study were run in parallel and assume a mutually-confirmative connotation. On the one hand, we survey 162 developers of the 9 considered systems to investigate whether developers perceive relationship between community smells and the code smells found in those projects. On the other hand, we perform a fine-grained analysis into the 117 releases of our dataset to measure the extent to which community smells impact code smell intensity (i.e., criticality). We then propose a code smell intensity prediction model that relies on both technical and community-related aspects. The results of both sides of our mixed-methods study lead to one conclusion: community-related factors contribute to the intensity of code smells. This conclusion supports the joint use of community and code smells detection as a mechanism for the joint management of technical and social problems around software development communities.

TEST SMELLS

An Empirical Analysis of the Distribution of Unit Test Smells and Their Impact on Software Maintenance

Gabriele Bavota[1], Abdallah Qusef[1], Rocco Oliveto[2], Andrea De Lucia[1], David Binkley[3]
[1]University of Salerno, Fisciano (SA), Italy
[2]University of Molise, Pesche (IS), Italy
[3]Loyola University Maryland, Baltimore, USA
gbavota@unisa.it, aqusef@unisa.it, rocco.oliveto@unimol.it, adelucia@unisa.it, binkley@...

An Empirical Investigation into the Nature of Test Smells

Michele Tufano[1], Fabio Palomba[2], Gabriele Bavota[3], Massimiliano Di Penta[4]
Rocco Oliveto[5], Andrea De Lucia[2], Denys Poshyvanyk[1]
[1] The College of William and Mary, USA — [2] University of Salerno, Italy — [3] Università della Svizzera italiana (USI), Switzerland — [4] University of Sannio, Italy — [5] University of Molise, Italy

Affect maintainability…

…but developers rarely perceive them as important

LINGUISTIC ANTIPATTERNS

Linguistic antipatterns: what they are and how developers perceive them

Venera Arnaoudova · Massimiliano Di Penta · Giuliano Antoniol

Published online: 29 January 2015

Abstract Antipatterns are known as poor solutions to recurring problems. For example, Brown et al. and Fowler define practices concerning poor design or implementation solutions. However, we know that the source code lexicon is part of the factors that affect the psychological complexity of a program, i.e., factors that make a program difficult to understand and maintain by humans. The aim of this work is to identify recurring poor practices related to inconsistencies among the naming, documentation, and implementation of an entity—called Linguistic Antipatterns (LAs)—that may impair program understanding. To

Perceived as serious concerns by developers

When present in APIs, correlate with the increase of Stack Overflow questions [Aghajani et al.]

SQL SMELLS

Tool for detecting query antipatterns from Bill Karwins SQL Antipatterns catalog

A Static Code Smell Detector for SQL Queries Embedded in Java Code

Csaba Nagy*, Anthony Cleve†
PReCISE Research Center, University of Namur, Belgium
*csaba.nagy@unamur.be, †anthony.cleve@unamur.be

Abstract—A database plays a central role in the architecture of an information system, and the way it stores the data delimits its main features. However, it is not just the data that matters. The way it is handled, i.e., how the application communicates with the database is of critical importance too. Therefore the implementation of such a communication layer has to be reliable and efficient. SQL is a popular language to query a database, and modern technologies rely on it (or its dialects) as query strings embedded in the application code. In many languages (e.g. in Java), an embedded query is typically constructed through several string operations that obstruct developers in

technologies to communicate with a DB, while JDBC occurs as the only database framework in 56.3% of the projects [3].
Database access technologies intend to help developers in various ways. They make it easier to integrate the communication with the database into the application code, e.g., by providing a link between Java classes and database entities (e.g. ORMs), or merely by supporting to reuse and construct queries (e.g. prepared statements). However, as a drawback, a developer hardly sees the final SQL query that is, in the

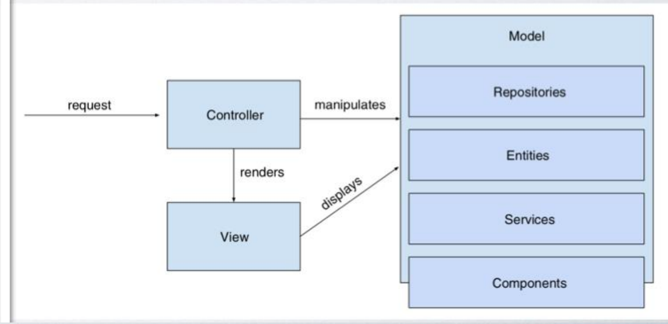
CODE SMELLS IN MODEL-VIEW CONTROLLER ARCHITECURES

Code smells for Model-View-Controller architectures

Maurício Aniche[1] · Gabriele Bavota[2] · Christoph Treude[3] · Marco Aurélio Gerosa[4] · Arie van Deursen[1]

Published online: 12 September 2017

Abstract Previous studies have shown the negative effects that low-quality code can have on maintainability proxies, such as code change- and defect-proneness. One of the symptoms of low-quality code are code smells, defined as sub-optimal implementation choices. While this definition is quite general and seems to suggest a wide spectrum of smells that can affect software systems, the research literature mostly focuses on the set of smells defined in



SERVICE ORIENTED ARCHITECTURE SMELLS

Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces

Mathieu Nayrolles, Naouel Moha, and Petko Valtchev

Investigating the Change-proneness of Service Patterns and Antipatterns

Francis Palma*†, Le An‡, Foutse Khomh‡, Naouel Moha† and Yann-Gaël Guéhéneuc*
*Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
†Latece, Département d'informatique, Université du Québec à Montréal, Canada
‡SWAT, DGIGL, École Polytechnique de Montréal, Canada
Email: {francis.palma, le.an, foutse.khomh, yann-gael.gueheneuc}@polymtl.ca, moha.naouel@uqam.ca

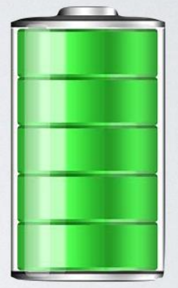
ENERGY SMELLS

470 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 44, NO. 5, MAY 2018

EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps

Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury

Abstract—Increased usage of mobile devices, such as smartphones and tablets, has led to widespread popularity and usage of mobile apps. If not carefully developed, such apps may demonstrate energy-inefficient behaviour, where one or more energy-intensive hardware components (such as Wifi, GPS, etc) are left in a high-power state, even when no apps are using these components. We refer to such kind of energy-inefficiencies as energy bugs. Executing an app with an energy bug causes the mobile device to exhibit poor energy consumption behaviour and a drastically shortened battery life. Since mobiles apps can have huge input domains, therefore exhaustive exploration is often impractical. We believe that there is a need for a framework that can systematically detect and fix energy bugs in mobile apps in a scalable fashion. To address this need, we have developed EnergyPatch, a framework that uses a combination of static and dynamic analysis techniques to detect, validate and repair energy bugs in Android apps. The use of a light-weight, static analysis technique enables EnergyPatch to quickly narrow down to the potential program paths along which energy bugs may occur. Subsequent exploration of these potentially buggy program paths using a dynamic analysis technique helps in validations of the reported bugs and to generate test cases. Finally, EnergyPatch generates repair expressions to fix the validated energy bugs. Evaluation with real-life apps from repositories such as F-droid and Github, shows that EnergyPatch is scalable and can produce results in reasonable amount of time. Additionally, we observed that the repair expressions generated by EnergyPatch could bring down the energy consumption on tested apps up to 60 percent.



INFRASTRUCTURE-AS-CODE SMELLS

Does Your Configuration Code Smell?

Tushar Sharma, Marios Fragkoulis and Diomidis Spinellis
Dept of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
{tushar,mfg,dds}@aueb.gr

ABSTRACT
Infrastructure as Code (IaC) is the practice of specifying computing system configurations through code, and managing them through traditional software engineering methods. The wide adoption of configuration management and increasing size and complexity of the associated code, prompt for assessing, maintaining, and improving the configuration code's quality. In this context, traditional software engineering knowledge and best practices associated with code quality management can be leveraged to assess and manage

1. INTRODUCTION
Infrastructure as Code (IaC) [13] is the practice of specifying computing system configurations through code, automating system deployment, and managing the system configurations through traditional software engineering methods. For example, a server farm that contains numerous nodes with different hardware configurations and different software package requirements can be specified using configuration management languages such as Puppet [39], Chef [37], CFEngine [4], or Ansible [1] and deployed automatically

SMELLS IN CONTINUOUS INTEGRATION PIPELINES

Use and Misuse of Continuous Integration Features

An Empirical Study of Projects that (mis)use Travis CI

Keheliya Gallaba, *Student Member, IEEE,* and Shane McIntosh, *Member, IEEE*

Automated Reporting of Anti-Patterns and Decay in Continuous Integration

Carmine Vassallo, *Department of Informatics, University of Zurich*, Zurich, Switzerland, vassallo@ifi.uzh.ch
Sebastian Proksch, *Department of Informatics, University of Zurich*, Zurich, Switzerland, proksch@ifi.uzh.ch
Harald C. Gall, *Department of Informatics, University of Zurich*, Zurich, Switzerland, gall@ifi.uzh.ch
Massimiliano Di Penta, *Department of Engineering, University of Sannio*, Benevento, Italy, dipenta@unisannio.it

COMMUNITY SMELLS

Beyond Technical Aspects: How Do Community Smells Influence the Intensity of Code Smells?

Fabio Palomba, *Member, IEEE,* Damian A. Tamburri, *Member, IEEE,* Francesca Arcelli Fontana, *Member, IEEE,* Rocco Oliveto, *Member, IEEE,* Andy Zaidman, *Member, IEEE,* Alexander Serebrenik, *Senior Member, IEEE.*

Abstract—Code smells are poor implementation choices applied by developers during software evolution that often lead to critical flaws or failure. Much in the same way, community smells reflect the presence of organizational and socio-technical issues within a software community that may lead to additional project costs. Recent empirical studies provide evidence that community smells are often—if not always—connected to circumstances such as code smells. In this paper we look deeper into this connection by conducting a mixed-methods empirical study of 117 releases from 9 open-source systems. The qualitative and quantitative sides of our mixed-methods study were run in parallel and assume a mutually-confirmative connotation. On the one hand, we survey 162 developers of the 9 considered systems to investigate whether developers perceive relationship between community smells and the code smells found in those projects. On the other hand, we perform a fine-grained analysis into the 117 releases of our dataset to measure the extent to which community smells impact code smell intensity (i.e., criticality). We then propose a code smell intensity prediction model that relies on both technical and community-related aspects. The results of both sides of our mixed-methods study lead to one conclusion: community-related factors contribute to the intensity of code smells. This conclusion supports the joint use of community and code smells detection as a mechanism for the joint management of technical and social problems around software development communities.

SQL SMELLS

Tool for detecting query antipatterns from Bill Karwins SQL

Similar change-, defect-proneness, and survival properties of traditional smells

PReCISE Research Center, University of Namur, Belgium
*csaba.nagy@unamur.be, †anthony.cleve@unamur.be

***Abstract*—A database plays a central role in the architecture of an information system, and the way it stores the data delimits its main features. However, it is not just the data that matters. The way it is handled, i.e., how the application communicates with the database is of critical importance too. Therefore the implementation of such a communication layer has to be reliable and efficient. SQL is a popular language to query a database, and modern technologies rely on it (or its dialects) as query strings embedded in the application code. In many languages (e.g. in Java), an embedded query is typically constructed through several string operations that obstruct developers in**

technologies to communicate with a DB, while JDBC occurs as the only database framework in 56.3% of the projects [3].

Database access technologies intend to help developers in various ways. They make it easier to integrate the communication with the database into the application code, e.g., by providing a link between Java classes and database entities (e.g. ORMs), or merely by supporting to reuse and construct queries (e.g. prepared statements). However, as a drawback, a developer hardly sees the final SQL query that is, in the

TEST SMELLS

An Empirical Analysis of the Distribution of Unit Test Smells and Their Impact on Software Maintenance

Gabriele Bavota[1], Abdallah Qusef[1], Rocco Oliveto[2], Andrea De Lucia[1], David Binkley[3]
[1]University of Salerno, Fisciano (SA), Italy
[2]University of Molise, Pesche (IS), Italy
[3]Loyola University Maryland, Baltimore, USA

An Empirical Investigation into the Nature of Test Smells

Michele Tufano[1], Fabio Palomba[2], Gabriele Bavota[3], Massimiliano Di Penta[4]
Rocco Oliveto[5], Andrea De Lucia[2], Denys Poshyvanyk[1]
[1] The College of William and Mary, USA — [2] University of Salerno, Italy — [3] Università della Svizzera italiana (USI), Switzerland — [4] University of Sannio, Italy — [5] University of Molise, Italy

ABSTRACT

Affect maintainability…

…but developers rarely perceive them as important

LINGUISTIC ANTIPATTERNS

Linguistic antipatterns: what they are and how developers perceive them

Venera Arnaoudova · Massimiliano Di Penta · Giuliano Antoniol

Published online: 29 January 2015

Abstract Antipatterns are known as poor solutions to recurring problems. For example, Brown et al. and Fowler define practices concerning poor design or implementation solutions. However, we know that the source code lexicon is part of the factors that affect the psychological complexity of a program, i.e., factors that make a program difficult to understand and maintain by humans. The aim of this work is to identify recurring poor practices related to inconsistencies among the naming, documentation, and implementation of an entity—called Linguistic Antipatterns (LAs)—that may impair program understanding.

Perceived as serious concerns by developers

When present in APIs, correlate with the increase of Stack Overflow questions [Aghajani et al.]

SQL SMELLS

Tool for detecting query antipatterns from Bill Karwins SQL Antipatterns catalog

A Static Code Smell Detector for SQL Queries Embedded in Java Code

Csaba Nagy*, Anthony Cleve†
PReCISE Research Center, University of Namur, Belgium
*csaba.nagy@unamur.be, †anthony.cleve@unamur.be

Abstract—**A database plays a central role in the architecture of an information system, and the way it stores the data delimits its main features. However, it is not just the data that matters. The way it is handled, i.e., how the application communicates with the database is of critical importance too. Therefore the implementation of such a communication layer has to be reliable and efficient. SQL is a popular language to query a database, and modern technologies rely on it (or its dialects) as query strings embedded in the application code. In many languages (e.g. in Java), an embedded query is typically constructed through several string operations that obstruct developers in**

technologies to communicate with a DB, while JDBC occurs as the only database framework in 56.3% of the projects [3].

Database access technologies intend to help developers in various ways. They make it easier to integrate the communication with the database into the application code, e.g., by providing a link between Java classes and database entities (e.g. ORMs), or merely by supporting to reuse and construct queries (e.g. prepared statements). However, as a drawback, a developer hardly sees the final SQL query that is, in the

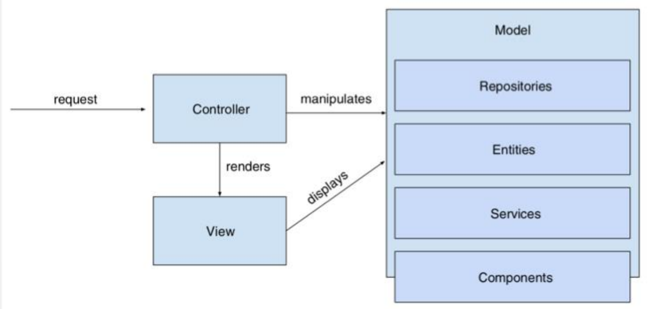
CODE SMELLS IN MODEL-VIEW CONTROLLER ARCHITECURES

Code smells for Model-View-Controller architectures

Maurício Aniche[1] · Gabriele Bavota[2] · Christoph Treude[3] · Marco Aurélio Gerosa[4] · Arie van Deursen[1]

Published online: 12 September 2017

Abstract Previous studies have shown the negative effects that low-quality code can have on maintainability proxies, such as code change- and defect-proneness. One of the symptoms of low-quality code are code smells, defined as sub-optimal implementation choices. While this definition is quite general and seems to suggest a wide spectrum of smells that can affect software systems, the research literature mostly focuses on the set of smells defined in



SERVICE ORIENTED ARCHITECTURE SMELLS

Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces

Mathieu Nayrolles, Naouel Moha, and Petko Valtchev

Investigating the Change-proneness of Service Patterns and Antipatterns

Francis Palma*†, Le An‡, Foutse Khomh‡, Naouel Moha† and Yann-Gaël Guéhéneuc*
*Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
†Latece, Département d'informatique, Université du Québec à Montréal, Canada
‡SWAT, DGIGL, École Polytechnique de Montréal, Canada
Email: {francis.palma, le.an, foutse.khomh, yann-gael.gueheneuc}@polymtl.ca, moha.naouel@uqam.ca

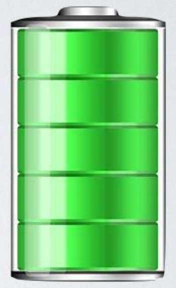
ENERGY SMELLS

470 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 44, NO. 5, MAY 2018

EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps

Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury

Abstract—Increased usage of mobile devices, such as smartphones and tablets, has led to widespread popularity and usage of mobile apps. If not carefully developed, such apps may demonstrate energy-inefficient behaviour, where one or more energy-intensive hardware components (such as Wifi, GPS, etc) are left in a high-power state, even when no apps are using these components. We refer to such kind of energy-inefficiencies as energy bugs. Executing an app with an energy bug causes the mobile device to exhibit poor energy consumption behaviour and a drastically shortened battery life. Since mobiles apps can have huge input domains, therefore exhaustive exploration is often impractical. We believe that there is a need for a framework that can systematically detect and fix energy bugs in mobile apps in a scalable fashion. To address this need, we have developed EnergyPatch, a framework that uses a combination of static and dynamic analysis techniques to detect, validate and repair energy bugs in Android apps. The use of a light-weight, static analysis technique enables EnergyPatch to quickly narrow down to the potential program paths along which energy bugs may occur. Subsequent exploration of these potentially buggy program paths using a dynamic analysis technique helps in validations of the reported bugs and to generate test cases. Finally, EnergyPatch generates repair expressions to fix the validated energy bugs. Evaluation with real-life apps from repositories such as F-droid and Github, shows that EnergyPatch is scalable and can produce results in reasonable amount of time. Additionally, we observed that the repair expressions generated by EnergyPatch could bring down the energy consumption on tested apps up to 60 percent.



INFRASTRUCTURE-AS-CODE SMELLS

Does Your Configuration Code Smell?

Tushar Sharma, Marios Fragkoulis and Diomidis Spinellis
Dept of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
{tushar,mfg,dds}@aueb.gr

ABSTRACT

Infrastructure as Code (IaC) is the practice of specifying computing system configurations through code, and managing them through traditional software engineering methods. The wide adoption of configuration management and increasing size and complexity of the associated code, prompt for assessing, maintaining, and improving the configuration code's quality. In this context, traditional software engineering knowledge and best practices associated with code quality management can be leveraged to assess and manage

1. INTRODUCTION

Infrastructure as Code (IaC) [13] is the practice of specifying computing system configurations through code, automating system deployment, and managing the system configurations through traditional software engineering methods. For example, a server farm that contains numerous nodes with different hardware configurations and different software package requirements can be specified using configuration management languages such as Puppet [39], Chef [37], CFEngine [4], or Ansible [1] and deployed automatically

SMELLS IN CONTINUOUS INTEGRATION PIPELINES

Use and Misuse of Continuous Integration Features

An Empirical Study of Projects that (mis)use Travis CI

Keheliya Gallaba, *Student Member, IEEE,* and Shane McIntosh, *Member, IEEE*

1 INTRODUCTION

Automated Reporting of Anti-Patterns and Decay in Continuous Integration

Carmine Vassallo	Sebastian Proksch	Harald C. Gall	Massimiliano Di Penta
Department of Informatics	*Department of Informatics*	*Department of Informatics*	*Department of Engineering*
University of Zurich	*University of Zurich*	*University of Zurich*	*University of Sannio*
Zurich, Switzerland	Zurich, Switzerland	Zurich, Switzerland	Benevento, Italy
vassallo@ifi.uzh.ch	proksch@ifi.uzh.ch	gall@ifi.uzh.ch	dipenta@unisannio.it

COMMUNITY SMELLS

Beyond Technical Aspects: How Do Community Smells Influence the Intensity of Code Smells?

Fabio Palomba, *Member, IEEE,* Damian A. Tamburri, *Member, IEEE,* Francesca Arcelli Fontana, *Member, IEEE,* Rocco Oliveto, *Member, IEEE,* Andy Zaidman, *Member, IEEE,* Alexander Serebrenik, *Senior Member, IEEE.*

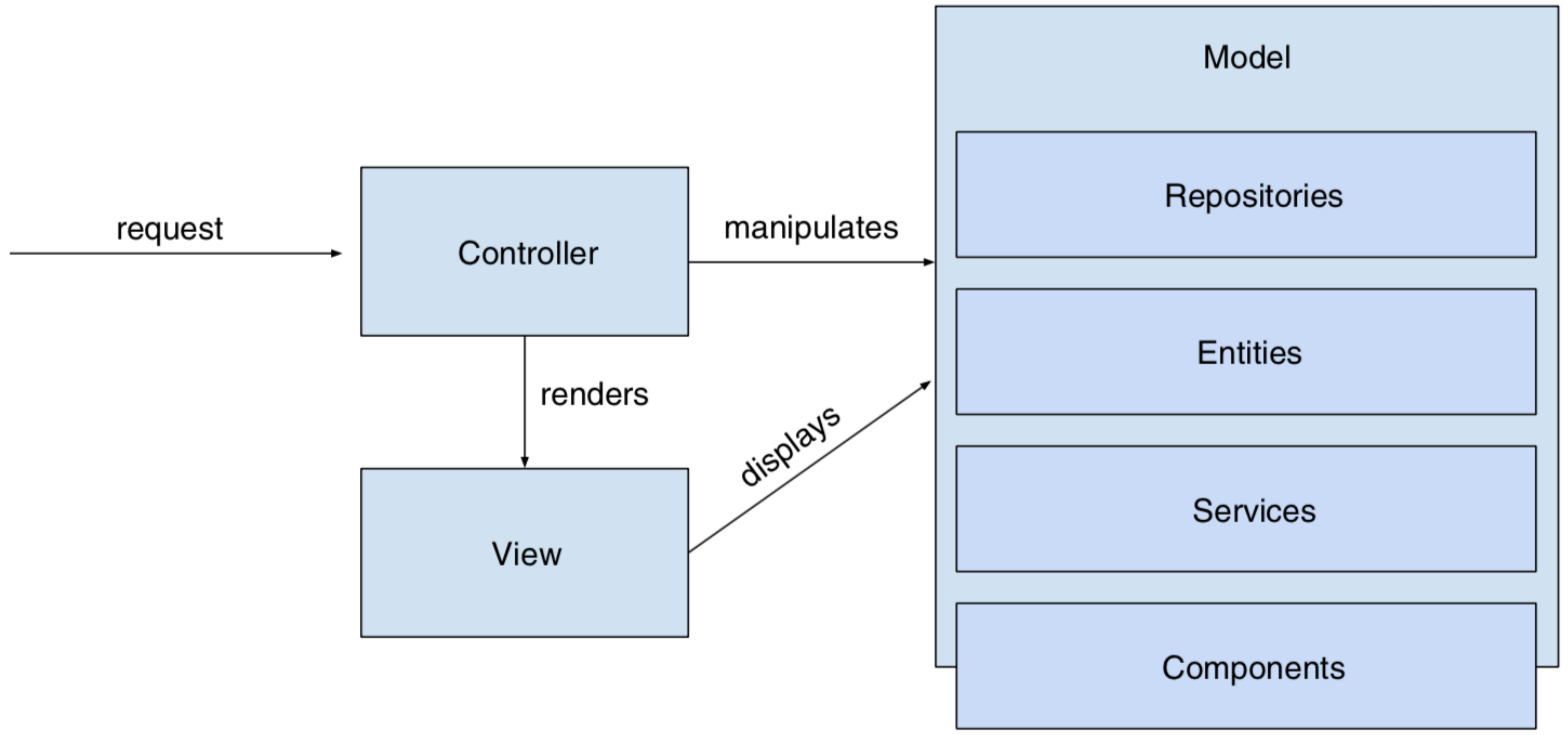
Abstract—Code smells are poor implementation choices applied by developers during software evolution that often lead to critical flaws or failure. Much in the same way, community smells reflect the presence of organizational and socio-technical issues within a software community that may lead to additional project costs. Recent empirical studies provide evidence that community smells are often—if not always—connected to circumstances such as code smells. In this paper we look deeper into this connection by conducting a mixed-methods empirical study of 117 releases from 9 open-source systems. The qualitative and quantitative sides of our mixed-methods study were run in parallel and assume a mutually-confirmative connotation. On the one hand, we survey 162 developers of the 9 considered systems to investigate whether developers perceive relationship between community smells and the code smells found in those projects. On the other hand, we perform a fine-grained analysis into the 117 releases of our dataset to measure the extent to which community smells impact code smell intensity (i.e., criticality). We then propose a code smell intensity prediction model that relies on both technical and community-related aspects. The results of both sides of our mixed-methods study lead to one conclusion: community-related factors contribute to the intensity of code smells. This conclusion supports the joint use of community and code smells detection as a mechanism for the joint management of technical and social problems around software development communities.

CODE SMELLS IN MODEL-VIEW CONTROLLER ARCHITECURES

Similar change-, defect-proneness, and survival properties of traditional smells

Published online: 12 September 2017

Abstract Previous studies have shown the negative effects that low-quality code can have on maintainability proxies, such as code change- and defect-proneness. One of the symptoms of low-quality code are code smells, defined as sub-optimal implementation choices. While this definition is quite general and seems to suggest a wide spectrum of smells that can affect software systems, the research literature mostly focuses on the set of smells defined in



TEST SMELLS

An Empirical Analysis of the Distribution of Unit Test Smells and Their Impact on Software Maintenance

Gabriele Bavota[1], Abdallah Qusef[1], Rocco Oliveto[2], Andrea De Lucia[1], David Binkley[3]
[1]University of Salerno, Fisciano (SA), Italy
[2]University of Molise, Pesche (IS), Italy
[3]Loyola University Maryland, Baltimore, USA
gbavota@unisa.it, aqusef@unisa.it, rocco.oliveto@unimol.it, adelucia@unisa.it, binkley@...

An Empirical Investigation into the Nature of Test Smells

Michele Tufano[1], Fabio Palomba[2], Gabriele Bavota[3], Massimiliano Di Penta[4]
Rocco Oliveto[5], Andrea De Lucia[2], Denys Poshyvanyk[1]
[1] The College of William and Mary, USA — [2] University of Salerno, Italy — [3] Università della Svizzera italiana (USI), Switzerland — [4] University of Sannio, Italy — [5] University of Molise, Italy

ABSTRACT
Test smells have been defined as poorly designed tests and, as reported by recent empirical studies, their presence may ...
of the system. To ease developers' burden in writing, organizing, and executing test suites, nowadays appropriate frameworks (e.g., JUnit [9])—conceived for unit testing but ...

Affect maintainability…

…but developers rarely perceive them as important

LINGUISTIC ANTIPATTERNS

Linguistic antipatterns: what they are and how developers perceive them

Venera Arnaoudova · Massimiliano Di Penta · Giuliano Antoniol

Published online: 29 January 2015

Abstract Antipatterns are known as poor solutions to recurring problems. For example, Brown et al. and Fowler define practices concerning poor design or implementation solutions. However, we know that the source code lexicon is part of the factors that affect the psychological complexity of a program, i.e., factors that make a program difficult to understand and maintain by humans. The aim of this work is to identify recurring poor practices related to inconsistencies among the naming, documentation, and implementation of an entity—called Linguistic Antipatterns (LAs)—that may impair program understanding. To ...

Perceived as serious concerns by developers

When present in APIs, correlate with the increase of Stack Overflow questions [Aghajani et al.]

SQL SMELLS

Tool for detecting query antipatterns from Bill Karwins SQL Antipatterns catalog

A Static Code Smell Detector for SQL Queries Embedded in Java Code

Csaba Nagy*, Anthony Cleve†
PReCISE Research Center, University of Namur, Belgium
*csaba.nagy@unamur.be, †anthony.cleve@unamur.be

Abstract—A database plays a central role in the architecture of an information system, and the way it stores the data delimits its main features. However, it is not just the data that matters. The way it is handled, i.e., how the application communicates with the database is of critical importance too. Therefore the implementation of such a communication layer has to be reliable and efficient. SQL is a popular language to query a database, and modern technologies rely on it (or its dialects) as query strings embedded in the application code. In many languages (e.g. in Java), an embedded query is typically constructed through several string operations that obstruct developers in ...

technologies to communicate with a DB, while JDBC occurs as the only database framework in 56.3% of the projects [3].
Database access technologies intend to help developers in various ways. They make it easier to integrate the communication with the database into the application code, e.g., by providing a link between Java classes and database entities (e.g. ORMs), or merely by supporting to reuse and construct queries (e.g. prepared statements). However, as a drawback, a developer hardly sees the final SQL query that is, in the ...

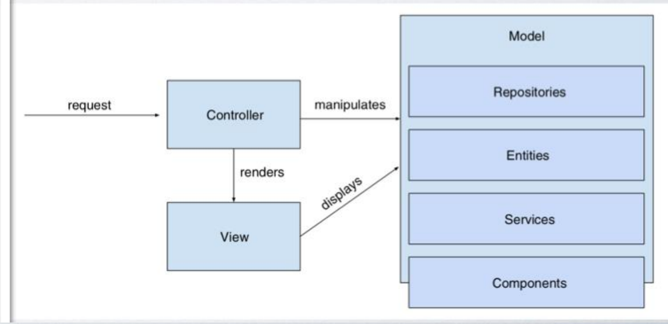
CODE SMELLS IN MODEL-VIEW CONTROLLER ARCHITECURES

Code smells for Model-View-Controller architectures

Maurício Aniche[1] · Gabriele Bavota[2] · Christoph Treude[3] · Marco Aurélio Gerosa[4] · Arie van Deursen[1]

Published online: 12 September 2017

Abstract Previous studies have shown the negative effects that low-quality code can have on maintainability proxies, such as code change- and defect-proneness. One of the symptoms of low-quality code are code smells, defined as sub-optimal implementation choices. While this definition is quite general and seems to suggest a wide spectrum of smells that can affect software systems, the research literature mostly focuses on the set of smells defined in ...



SERVICE ORIENTED ARCHITECTURE SMELLS

Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces

Mathieu Nayrolles, Naouel Moha, and Petko Valtchev
LATECE Team, Département d'informatique, Université du Québec à Montréal, Canada

Investigating the Change-proneness of Service Patterns and Antipatterns

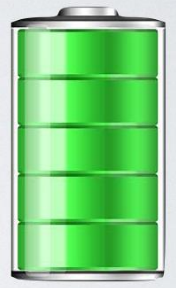
Francis Palma*†, Le An‡, Foutse Khomh‡, Naouel Moha† and Yann-Gaël Guéhéneuc*
*Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
†Latece, Département d'informatique, Université du Québec à Montréal, Canada
‡SWAT, DGIGL, École Polytechnique de Montréal, Canada
Email: {francis.palma, le.an, foutse.khomh, yann-gael.gueheneuc}@polymtl.ca, moha.naouel@uqam.ca

ENERGY SMELLS

EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps

Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury

Abstract—Increased usage of mobile devices, such as smartphones and tablets, has led to widespread popularity and usage of mobile apps. If not carefully developed, such apps may demonstrate energy-inefficient behaviour, where one or more energy-intensive hardware components (such as Wifi, GPS, etc) are left in a high-power state, even when no apps are using these components. We refer to such kind of energy-inefficiencies as energy bugs. Executing an app with an energy bug causes the mobile device to exhibit poor energy consumption behaviour and a drastically shortened battery life. Since mobiles apps can have huge input domains, therefore exhaustive exploration is often impractical. We believe that there is a need for a framework that can systematically detect and fix energy bugs in mobile apps in a scalable fashion. To address this need, we have developed EnergyPatch, a framework that uses a combination of static and dynamic analysis techniques to detect, validate and repair energy bugs in Android apps. The use of a light-weight, static analysis technique enables EnergyPatch to quickly narrow down to the potential program paths along which energy bugs may occur. Subsequent exploration of these potentially buggy program paths using a dynamic analysis technique helps in validations of the reported bugs and to generate test cases. Finally, EnergyPatch generates repair expressions to fix the validated energy bugs. Evaluation with real-life apps from repositories such as F-droid and Github, shows that EnergyPatch is scalable and can produce results in reasonable amount of time. Additionally, we observed that the repair expressions generated by EnergyPatch could bring down the energy consumption on tested apps up to 60 percent.



INFRASTRUCTURE-AS-CODE SMELLS

Does Your Configuration Code Smell?

Tushar Sharma, Marios Fragkoulis and Diomidis Spinellis
Dept of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
{tushar,mfg,dds}@aueb.gr

ABSTRACT
Infrastructure as Code (IaC) is the practice of specifying computing system configurations through code, and managing them through traditional software engineering methods. The wide adoption of configuration management and increasing size and complexity of the associated code, prompt for assessing, maintaining, and improving the configuration code's quality. In this context, traditional software engineering knowledge and best practices associated with code quality management can be leveraged to assess and manage ...

1. INTRODUCTION
Infrastructure as Code (IaC) [13] is the practice of specifying computing system configurations through code, automating system deployment, and managing the system configurations through traditional software engineering methods. For example, a server farm that contains numerous nodes with different hardware configurations and different software package requirements can be specified using configuration management languages such as Puppet [39], Chef [37], CFEngine [4], or Ansible [1] and deployed automatically ...

SMELLS IN CONTINUOUS INTEGRATION PIPELINES

Use and Misuse of Continuous Integration Features

An Empirical Study of Projects that (mis)use Travis CI

Keheliya Gallaba, *Student Member, IEEE*, and Shane McIntosh, *Member, IEEE*

Automated Reporting of Anti-Patterns and Decay in Continuous Integration

Carmine Vassallo, *Department of Informatics, University of Zurich*, Zurich, Switzerland, vassallo@ifi.uzh.ch
Sebastian Proksch, *Department of Informatics, University of Zurich*, Zurich, Switzerland, proksch@ifi.uzh.ch
Harald C. Gall, *Department of Informatics, University of Zurich*, Zurich, Switzerland, gall@ifi.uzh.ch
Massimiliano Di Penta, *Department of Engineering, University of Sannio*, Benevento, Italy, dipenta@unisannio.it

COMMUNITY SMELLS

Beyond Technical Aspects: How Do Community Smells Influence the Intensity of Code Smells?

Fabio Palomba, *Member, IEEE*, Damian A. Tamburri, *Member, IEEE*, Francesca Arcelli Fontana, *Member, IEEE*, Rocco Oliveto, *Member, IEEE*, Andy Zaidman, *Member, IEEE*, Alexander Serebrenik, *Senior Member, IEEE*.

Abstract—Code smells are poor implementation choices applied by developers during software evolution that often lead to critical flaws or failure. Much in the same way, community smells reflect the presence of organizational and socio-technical issues within a software community that may lead to additional project costs. Recent empirical studies provide evidence that community smells are often—if not always—connected to circumstances such as code smells. In this paper we look deeper into this connection by conducting a mixed-methods empirical study of 117 releases from 9 open-source systems. The qualitative and quantitative sides of our mixed-methods study were run in parallel and assume a mutually-confirmative connotation. On the one hand, we survey 162 developers of the 9 considered systems to investigate whether developers perceive relationship between community smells and the code smells found in those projects. On the other hand, we perform a fine-grained analysis into the 117 releases of our dataset to measure the extent to which community smells impact code smell intensity (i.e., criticality). We then propose a code smell intensity prediction model that relies on both technical and community-related aspects. The results of both sides of our mixed-methods study lead to one conclusion: community-related factors contribute to the intensity of code smells. This conclusion supports the joint use of community and code smells detection as a mechanism for the joint management of technical and social problems around software development communities.

SERVICE ORIENTED ARCHITECTURE SMELLS

Service antipatterns are more change-prone and require more maintenance effort than other services

Investigating the Change-proneness of Service Patterns and Antipatterns

Francis Palma*†, Le An‡, Foutse Khomh‡, Naouel Moha† and Yann-Gaël Guéhéneuc*
*Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
†Latece, Département d'informatique, Université du Québec à Montréal, Canada
‡SWAT, DGIGL, École Polytechnique de Montréal, Canada
Email: {francis.palma, le.an, foutse.khomh, yann-gael.gueheneuc}@polymtl.ca, moha.naouel@uqam.ca

Abstract—Servic
systems, evolve du
execution contexts.
the design and red
may result in poo
antipatterns. SOA
and reusability of
and then remove t
detection are still i
for their automati
and innovative ap
SOMAD (Service
which is an evoluti

TEST SMELLS

An Empirical Analysis of the Distribution of Unit Test Smells and Their Impact on Software Maintenance

Gabriele Bavota[1], Abdallah Qusef[1], Rocco Oliveto[2], Andrea De Lucia[1], David Binkley[3]
[1]University of Salerno, Fisciano (SA), Italy
[2]University of Molise, Pesche (IS), Italy
[3]Loyola University Maryland, Baltimore, USA

An Empirical Investigation into the Nature of Test Smells

Michele Tufano[1], Fabio Palomba[2], Gabriele Bavota[3], Massimiliano Di Penta[4]
Rocco Oliveto[5], Andrea De Lucia[2], Denys Poshyvanyk[1]
[1] The College of William and Mary, USA — [2] University of Salerno, Italy — [3] Università della Svizzera italiana (USI), Switzerland — [4] University of Sannio, Italy — [5] University of Molise, Italy

ABSTRACT

Affect maintainability…

…but developers rarely perceive them as important

LINGUISTIC ANTIPATTERNS

Linguistic antipatterns: what they are and how developers perceive them

Venera Arnaoudova · Massimiliano Di Penta · Giuliano Antoniol

Published online: 29 January 2015

Abstract Antipatterns are known as poor solutions to recurring problems. For example, Brown et al. and Fowler define practices concerning poor design or implementation solutions. However, we know that the source code lexicon is part of the factors that affect the psychological complexity of a program, i.e., factors that make a program difficult to understand and maintain by humans. The aim of this work is to identify recurring poor practices related to inconsistencies among the naming, documentation, and implementation of an entity—called Linguistic Antipatterns (LAs)—that may impair program understanding. To

Perceived as serious concerns by developers

When present in APIs, correlate with the increase of Stack Overflow questions [Aghajani et al.]

SQL SMELLS

Tool for detecting query antipatterns from Bill Karwins SQL Antipatterns catalog

A Static Code Smell Detector for SQL Queries Embedded in Java Code

Csaba Nagy*, Anthony Cleve†
PReCISE Research Center, University of Namur, Belgium
*csaba.nagy@unamur.be, †anthony.cleve@unamur.be

Abstract—A database plays a central role in the architecture of an information system, and the way it stores the data delimits its main features. However, it is not just the data that matters. The way it is handled, i.e., how the application communicates with the database is of critical importance too. Therefore the implementation of such a communication layer has to be reliable and efficient. SQL is a popular language to query a database, and modern technologies rely on it (or its dialects) as query strings embedded in the application code. In many languages (e.g. in Java), an embedded query is typically constructed through several string operations that obstruct developers in

technologies to communicate with a DB, while JDBC occurs as the only database framework in 56.3% of the projects [3].

Database access technologies intend to help developers in various ways. They make it easier to integrate the communication with the database into the application code, e.g., by providing a link between Java classes and database entities (e.g. ORMs), or merely by supporting to reuse and construct queries (e.g. prepared statements). However, as a drawback, a developer hardly sees the final SQL query that is, in the

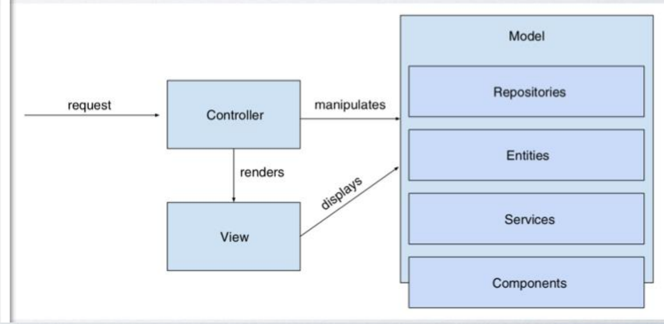
CODE SMELLS IN MODEL-VIEW CONTROLLER ARCHITECURES

Code smells for Model-View-Controller architectures

Maurício Aniche[1] · Gabriele Bavota[2] · Christoph Treude[3] · Marco Aurélio Gerosa[4] · Arie van Deursen[1]

Published online: 12 September 2017

Abstract Previous studies have shown the negative effects that low-quality code can have on maintainability proxies, such as code change- and defect-proneness. One of the symptoms of low-quality code are code smells, defined as sub-optimal implementation choices. While this definition is quite general and seems to suggest a wide spectrum of smells that can affect software systems, the research literature mostly focuses on the set of smells defined in



SERVICE ORIENTED ARCHITECTURE SMELLS

Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces

Mathieu Nayrolles, Naouel Moha, and Petko Valtchev

Investigating the Change-proneness of Service Patterns and Antipatterns

Francis Palma*†, Le An‡, Foutse Khomh‡, Naouel Moha† and Yann-Gaël Guéhéneuc*
*Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
†Latece, Département d'informatique, Université du Québec à Montréal, Canada
‡SWAT, DGIGL, École Polytechnique de Montréal, Canada
Email: {francis.palma, le.an, foutse.khomh, yann-gael.gueheneuc}@polymtl.ca, moha.naouel@uqam.ca

ENERGY SMELLS

470 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 44, NO. 5, MAY 2018

EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps

Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury

Abstract—Increased usage of mobile devices, such as smartphones and tablets, has led to widespread popularity and usage of mobile apps. If not carefully developed, such apps may demonstrate energy-inefficient behaviour, where one or more energy-intensive hardware components (such as Wifi, GPS, etc) are left in a high-power state, even when no apps are using these components. We refer to such kind of energy-inefficiencies as energy bugs. Executing an app with an energy bug causes the mobile device to exhibit poor energy consumption behaviour and a drastically shortened battery life. Since mobiles apps can have huge input domains, therefore exhaustive exploration is often impractical. We believe that there is a need for a framework that can systematically detect and fix energy bugs in mobile apps in a scalable fashion. To address this need, we have developed EnergyPatch, a framework that uses a combination of static and dynamic analysis techniques to detect, validate and repair energy bugs in Android apps. The use of a light-weight, static analysis technique enables EnergyPatch to quickly narrow down to the potential program paths along which energy bugs may occur. Subsequent exploration of these potentially buggy program paths using a dynamic analysis technique helps in validations of the reported bugs and to generate test cases. Finally, EnergyPatch generates repair expressions to fix the validated energy bugs. Evaluation with real-life apps from repositories such as F-droid and Github, shows that EnergyPatch is scalable and can produce results in reasonable amount of time. Additionally, we observed that the repair expressions generated by EnergyPatch could bring down the energy consumption on tested apps up to 60 percent.

INFRASTRUCTURE-AS-CODE SMELLS

Does Your Configuration Code Smell?

Tushar Sharma, Marios Fragkoulis and Diomidis Spinellis
Dept of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
{tushar,mfg,dds}@aueb.gr

ABSTRACT

Infrastructure as Code (IaC) is the practice of specifying computing system configurations through code, and managing them through traditional software engineering methods. The wide adoption of configuration management and increasing size and complexity of the associated code, prompt for assessing, maintaining, and improving the configuration code's quality. In this context, traditional software engineering knowledge and best practices associated with code quality management can be leveraged to assess and manage

1. INTRODUCTION

Infrastructure as Code (IaC) [13] is the practice of specifying computing system configurations through code, automating system deployment, and managing the system configurations through traditional software engineering methods. For example, a server farm that contains numerous nodes with different hardware configurations and different software package requirements can be specified using configuration management languages such as Puppet [39], Chef [37], CFEngine [4], or Ansible [1] and deployed automatically

SMELLS IN CONTINUOUS INTEGRATION PIPELINES

Use and Misuse of Continuous Integration Features

An Empirical Study of Projects that (mis)use Travis CI

Keheliya Gallaba, *Student Member, IEEE*, and Shane McIntosh, *Member, IEEE*

Automated Reporting of Anti-Patterns and Decay in Continuous Integration

Carmine Vassallo, *Department of Informatics, University of Zurich*, Zurich, Switzerland, vassallo@ifi.uzh.ch
Sebastian Proksch, *Department of Informatics, University of Zurich*, Zurich, Switzerland, proksch@ifi.uzh.ch
Harald C. Gall, *Department of Informatics, University of Zurich*, Zurich, Switzerland, gall@ifi.uzh.ch
Massimiliano Di Penta, *Department of Engineering, University of Sannio*, Benevento, Italy, dipenta@unisannio.it

COMMUNITY SMELLS

Beyond Technical Aspects: How Do Community Smells Influence the Intensity of Code Smells?

Fabio Palomba, *Member, IEEE*, Damian A. Tamburri, *Member, IEEE*, Francesca Arcelli Fontana, *Member, IEEE*, Rocco Oliveto, *Member, IEEE*, Andy Zaidman, *Member, IEEE*, Alexander Serebrenik, *Senior Member, IEEE*.

Abstract—Code smells are poor implementation choices applied by developers during software evolution that often lead to critical flaws or failure. Much in the same way, community smells reflect the presence of organizational and socio-technical issues within a software community that may lead to additional project costs. Recent empirical studies provide evidence that community smells are often—if not always—connected to circumstances such as code smells. In this paper we look deeper into this connection by conducting a mixed-methods empirical study of 117 releases from 9 open-source systems. The qualitative and quantitative sides of our mixed-methods study were run in parallel and assume a mutually-confirmative connotation. On the one hand, we survey 162 developers of the 9 considered systems to investigate whether developers perceive relationship between community smells and the code smells found in those projects. On the other hand, we perform a fine-grained analysis into the 117 releases of our dataset to measure the extent to which community smells impact code smell intensity (i.e., criticality). We then propose a code smell intensity prediction model that relies on both technical and community-related aspects. The results of both sides of our mixed-methods study lead to one conclusion: community-related factors contribute to the intensity of code smells. This conclusion supports the joint use of community and code smells detection as a mechanism for the joint management of technical and social problems around software development communities.

ENERGY SMELLS

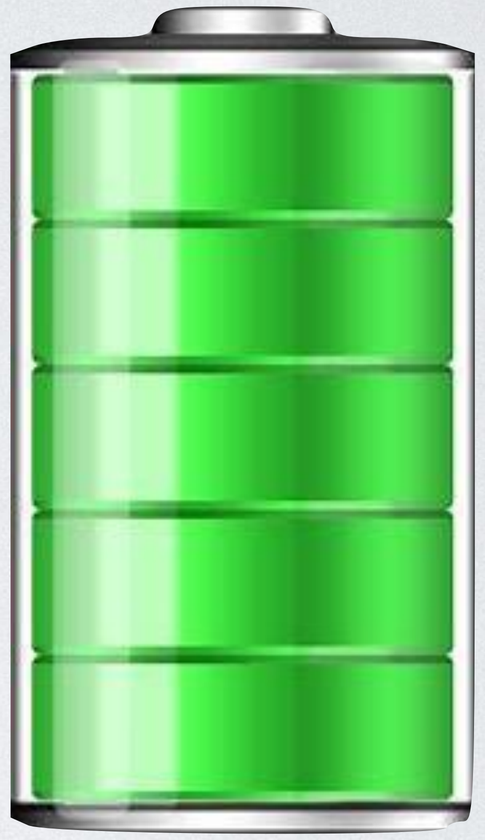
470 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 44, NO. 5, MAY 2018

EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps

Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury

Abstract—Increased usage of mobile devices, such as smartphones and tablets, has led to widespread popularity and usage of

Removing energy smells can reduce energy consumption of apps by up to 60 percent.



TEST SMELLS

An Empirical Analysis of the Distribution of Unit Test Smells and Their Impact on Software Maintenance

Gabriele Bavota[1], Abdallah Qusef[1], Rocco Oliveto[2], Andrea De Lucia[1], David Binkley[3]
[1]University of Salerno, Fisciano (SA), Italy
[2]University of Molise, Pesche (IS), Italy
[3]Loyola University Maryland, Baltimore, USA
gbavota@unisa.it, aqusef@unisa.it, rocco.oliveto@unimol.it, adelucia@unisa.it, binkley@...

An Empirical Investigation into the Nature of Test Smells

Michele Tufano[1], Fabio Palomba[2], Gabriele Bavota[3], Massimiliano Di Penta[4]
Rocco Oliveto[5], Andrea De Lucia[2], Denys Poshyvanyk[1]
[1] The College of William and Mary, USA — [2] University of Salerno, Italy — [3] Università della Svizzera italiana (USI), Switzerland — [4] University of Sannio, Italy — [5] University of Molise, Italy

ABSTRACT
Test smells have been defined as poorly designed tests and, as reported by recent empirical studies, their presence may ...
of the system. To ease developers' burden in writing, organizing, and executing test suites, nowadays appropriate frameworks (e.g., JUnit [9])—conceived for unit testing but ...

Affect maintainability…

…but developers rarely perceive them as important

LINGUISTIC ANTIPATTERNS

Linguistic antipatterns: what they are and how developers perceive them

Venera Arnaoudova · Massimiliano Di Penta · Giuliano Antoniol

Published online: 29 January 2015
© Springer Science+Business Media New York 2015

Abstract Antipatterns are known as poor solutions to recurring problems. For example, Brown et al. and Fowler define practices concerning poor design or implementation solutions. However, we know that the source code lexicon is part of the factors that affect the psychological complexity of a program, i.e., factors that make a program difficult to understand and maintain by humans. The aim of this work is to identify recurring poor practices related to inconsistencies among the naming, documentation, and implementation of an entity—called Linguistic Antipatterns (LAs)—that may impair program understanding. To

Perceived as serious concerns by developers

When present in APIs, correlate with the increase of Stack Overflow questions [Aghajani et al.]

SQL SMELLS

Tool for detecting query antipatterns from Bill Karwins SQL Antipatterns catalog

A Static Code Smell Detector for SQL Queries Embedded in Java Code

Csaba Nagy*, Anthony Cleve†
PReCISE Research Center, University of Namur, Belgium
*csaba.nagy@unamur.be, †anthony.cleve@unamur.be

Abstract—A database plays a central role in the architecture of an information system, and the way it stores the data delimits its main features. However, it is not just the data that matters. The way it is handled, i.e., how the application communicates with the database is of critical importance too. Therefore the implementation of such a communication layer has to be reliable and efficient. SQL is a popular language to query a database, and modern technologies rely on it (or its dialects) as query strings embedded in the application code. In many languages (e.g. in Java), an embedded query is typically constructed through several string operations that obstruct developers in ...

technologies to communicate with a DB, while JDBC occurs as the only database framework in 56.3% of the projects [3].
Database access technologies intend to help developers in various ways. They make it easier to integrate the communication with the database into the application code, e.g., by providing a link between Java classes and database entities (e.g. ORMs), or merely by supporting to reuse and construct queries (e.g. prepared statements). However, as a drawback, a developer hardly sees the final SQL query that is, in the ...

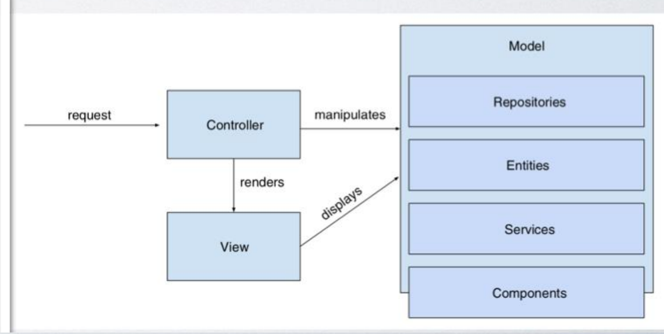
CODE SMELLS IN MODEL-VIEW CONTROLLER ARCHITECURES

Code smells for Model-View-Controller architectures

Maurício Aniche[1] · Gabriele Bavota[2] · Christoph Treude[3] · Marco Aurélio Gerosa[4] · Arie van Deursen[1]

Published online: 12 September 2017
© The Author(s) 2017. This article is an open access publication

Abstract Previous studies have shown the negative effects that low-quality code can have on maintainability proxies, such as code change- and defect-proneness. One of the symptoms of low-quality code are code smells, defined as sub-optimal implementation choices. While this definition is quite general and seems to suggest a wide spectrum of smells that can affect software systems, the research literature mostly focuses on the set of smells defined in



SERVICE ORIENTED ARCHITECTURE SMELLS

Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces

Mathieu Nayrolles, Naouel Moha, and Petko Valtchev

Abstract—Servi... systems, evolve du... execution contexts... the design and red... may result in poor... antipatterns. SOA... and reusability of... and then remove t... detection are still i... for their automati... and innovative ap... SOMAD (Service ... which is an evoluti...

Investigating the Change-proneness of Service Patterns and Antipatterns

Francis Palma*†, Le An‡, Foutse Khomh‡, Naouel Moha† and Yann-Gaël Guéhéneuc*
*Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
†Latece, Département d'informatique, Université du Québec à Montréal, Canada
‡SWAT, DGIGL, École Polytechnique de Montréal, Canada
Email: {francis.palma, le.an, foutse.khomh, yann-gael.gueheneuc}@polymtl.ca, moha.naouel@uqam.ca

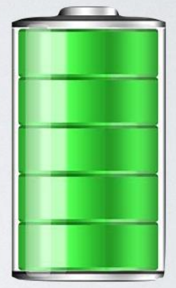
ENERGY SMELLS

470 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 44, NO. 5, MAY 2018

EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps

Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury

Abstract—Increased usage of mobile devices, such as smartphones and tablets, has led to widespread popularity and usage of mobile apps. If not carefully developed, such apps may demonstrate energy-inefficient behaviour, where one or more energy-intensive hardware components (such as Wifi, GPS, etc) are left in a high-power state, even when no apps are using these components. We refer to such kind of energy-inefficiencies as energy bugs. Executing an app with an energy bug causes the mobile device to exhibit poor energy consumption behaviour and a drastically shortened battery life. Since mobile apps can have huge input domains, therefore exhaustive exploration is often impractical. We believe that there is a need for a framework that can systematically detect and fix energy bugs in mobile apps in a scalable fashion. To address this need, we have developed EnergyPatch, a framework that uses a combination of static and dynamic analysis techniques to detect, validate and repair energy bugs in Android apps. The use of a light-weight, static analysis technique enables EnergyPatch to quickly narrow down to the potential program paths along which energy bugs may occur. Subsequent exploration of these potentially buggy program paths using a dynamic analysis technique helps in validations of the reported bugs and to generate test cases. Finally, EnergyPatch generates repair expressions to fix the validated energy bugs. Evaluation with real-life apps from repositories such as F-droid and Github, shows that EnergyPatch is scalable and can produce results in reasonable amount of time. Additionally, we observed that the repair expressions generated by EnergyPatch could bring down the energy consumption on tested apps up to 60 percent.



INFRASTRUCTURE-AS-CODE SMELLS

Does Your Configuration Code Smell?

Tushar Sharma, Marios Fragkoulis and Diomidis Spinellis
Dept of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
{tushar,mfg,dds}@aueb.gr

ABSTRACT
Infrastructure as Code (IaC) is the practice of specifying computing system configurations through code, and managing them through traditional software engineering methods. The wide adoption of configuration management and increasing size and complexity of the associated code, prompt for assessing, maintaining, and improving the configuration code's quality. In this context, traditional software engineering knowledge and best practices associated with code quality management can be leveraged to assess and manage ...

1. INTRODUCTION
Infrastructure as Code (IaC) [13] is the practice of specifying computing system configurations through code, automating system deployment, and managing the system configurations through traditional software engineering methods. For example, a server farm that contains numerous nodes with different hardware configurations and different software package requirements can be specified using configuration management languages such as Puppet [39], Chef [37], CFEngine [4], or Ansible [1] and deployed automatically without human intervention. ...

SMELLS IN CONTINUOUS INTEGRATION PIPELINES

Use and Misuse of Continuous Integration Features
An Empirical Study of Projects that (mis)use Travis CI

Keheliya Gallaba, *Student Member, IEEE*, and Shane McIntosh, *Member, IEEE*

Abstract—Continuous Integration (... appear in the version control system... there are several service providers... paper, we present a study of feature... are adopted by projects reveals that... associated with configuring job pro... TRAVIS CI specifications. We define... projects in the corpus (9.60%), and... anti-pattern removal tool for TRAVIS... automatically. Using GRETEL, we ha...

Index Terms—Continuous integrati...

1 INTRODUCTION

CONTINUOUS Integration (CI) is ... practice in which the latest co... downloaded onto dedicated mach...

Automated Reporting of Anti-Patterns and Decay in Continuous Integration

Carmine Vassallo, *Department of Informatics, University of Zurich*, Zurich, Switzerland, vassallo@ifi.uzh.ch
Sebastian Proksch, *Department of Informatics, University of Zurich*, Zurich, Switzerland, proksch@ifi.uzh.ch
Harald C. Gall, *Department of Informatics, University of Zurich*, Zurich, Switzerland, gall@ifi.uzh.ch
Massimiliano Di Penta, *Department of Engineering, University of Sannio*, Benevento, Italy, dipenta@unisannio.it

COMMUNITY SMELLS

Beyond Technical Aspects: How Do Community Smells Influence the Intensity of Code Smells?

Fabio Palomba, *Member, IEEE*, Damian A. Tamburri, *Member, IEEE*,
Francesca Arcelli Fontana, *Member, IEEE*, Rocco Oliveto, *Member, IEEE*,
Andy Zaidman, *Member, IEEE*, Alexander Serebrenik, *Senior Member, IEEE*.

Abstract—Code smells are poor implementation choices applied by developers during software evolution that often lead to critical flaws or failure. Much in the same way, community smells reflect the presence of organizational and socio-technical issues within a software community that may lead to additional project costs. Recent empirical studies provide evidence that community smells are often—if not always—connected to circumstances such as code smells. In this paper we look deeper into this connection by conducting a mixed-methods empirical study of 117 releases from 9 open-source systems. The qualitative and quantitative sides of our mixed-methods study were run in parallel and assume a mutually-confirmative connotation. On the one hand, we survey 162 developers of the 9 considered systems to investigate whether developers perceive relationship between community smells and the code smells found in those projects. On the other hand, we perform a fine-grained analysis into the 117 releases of our dataset to measure the extent to which community smells impact code smell intensity (i.e., criticality). We then propose a code smell intensity prediction model that relies on both technical and community-related aspects. The results of both sides of our mixed-methods study lead to one conclusion: community-related factors contribute to the intensity of code smells. This conclusion supports the joint use of community and code smells detection as a mechanism for the joint management of technical and social problems around software development communities.

TEST SMELLS

An Empirical Analysis of the Distribution of Unit Test Smells and Their Impact on Software Maintenance

Gabriele Bavota[1], Abdallah Qusef[1], Rocco Oliveto[2], Andrea De Lucia[1], David Binkley[3]
[1]University of Salerno, Fisciano (SA), Italy
[2]University of Molise, Pesche (IS), Italy
[3]Loyola University Maryland, Baltimore, USA

An Empirical Investigation into the Nature of Test Smells

Michele Tufano[1], Fabio Palomba[2], Gabriele Bavota[3], Massimiliano Di Penta[4], Rocco Oliveto[5], Andrea De Lucia[2], Denys Poshyvanyk[1]
[1] The College of William and Mary, USA — [2] University of Salerno, Italy — [3] Università della Svizzera italiana (USI), Switzerland — [4] University of Sannio, Italy — [5] University of Molise, Italy

Affect maintainability…

…but developers rarely perceive them as important

LINGUISTIC ANTIPATTERNS

Linguistic antipatterns: what they are and how developers perceive them

Venera Arnaoudova · Massimiliano Di Penta · Giuliano Antoniol

Published online: 29 January 2015

Abstract Antipatterns are known as poor solutions to recurring problems. For example, Brown et al. and Fowler define practices concerning poor design or implementation solutions. However, we know that the source code lexicon is part of the factors that affect the psychological complexity of a program, i.e., factors that make a program difficult to understand and maintain by humans. The aim of this work is to identify recurring poor practices related to inconsistencies among the naming, documentation, and implementation of an entity—called Linguistic Antipatterns (LAs)—that may impair program understanding. To

Perceived as serious concerns by developers

When present in APIs, correlate with the increase of Stack Overflow questions [Aghajani et al.]

SQL SMELLS

Tool for detecting query antipatterns from Bill Karwins SQL Antipatterns catalog

A Static Code Smell Detector for SQL Queries Embedded in Java Code

Csaba Nagy*, Anthony Cleve†
PReCISE Research Center, University of Namur, Belgium
*csaba.nagy@unamur.be, †anthony.cleve@unamur.be

Abstract—A database plays a central role in the architecture of an information system, and the way it stores the data delimits its main features. However, it is not just the data that matters. The way it is handled, i.e., how the application communicates with the database is of critical importance too. Therefore the implementation of such a communication layer has to be reliable and efficient. SQL is a popular language to query a database, and modern technologies rely on it (or its dialects) as query strings embedded in the application code. In many languages (e.g. in Java), an embedded query is typically constructed through several string operations that obstruct developers in

technologies to communicate with a DB, while JDBC occurs as the only database framework in 56.3% of the projects [3].

Database access technologies intend to help developers in various ways. They make it easier to integrate the communication with the database into the application code, e.g., by providing a link between Java classes and database entities (e.g. ORMs), or merely by supporting to reuse and construct queries (e.g. prepared statements). However, as a drawback, a developer hardly sees the final SQL query that is, in the

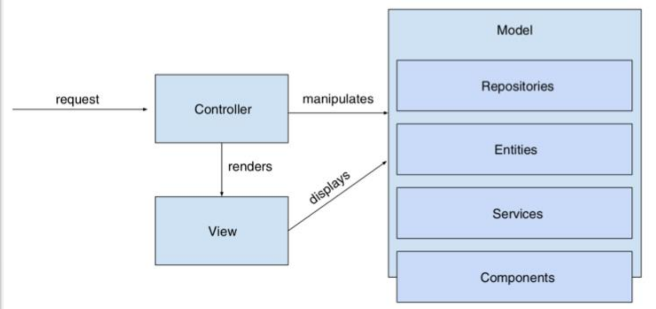
CODE SMELLS IN MODEL-VIEW CONTROLLER ARCHITECURES

Code smells for Model-View-Controller architectures

Maurício Aniche[1] · Gabriele Bavota[2] · Christoph Treude[3] · Marco Aurélio Gerosa[4] · Arie van Deursen[1]

Published online: 12 September 2017

Abstract Previous studies have shown the negative effects that low-quality code can have on maintainability proxies, such as code change- and defect-proneness. One of the symptoms of low-quality code are code smells, defined as sub-optimal implementation choices. While this definition is quite general and seems to suggest a wide spectrum of smells that can affect software systems, the research literature mostly focuses on the set of smells defined in



SERVICE ORIENTED ARCHITECTURE SMELLS

Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces

Mathieu Nayrolles, Naouel Moha, and Petko Valtchev

Investigating the Change-proneness of Service Patterns and Antipatterns

Francis Palma*†, Le An‡, Foutse Khomh‡, Naouel Moha† and Yann-Gaël Guéhéneuc*
*Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
†Latece, Département d'informatique, Université du Québec à Montréal, Canada
‡SWAT, DGIGL, École Polytechnique de Montréal, Canada
Email: {francis.palma, le.an, foutse.khomh, yann-gael.gueheneuc}@polymtl.ca, moha.naouel@uqam.ca

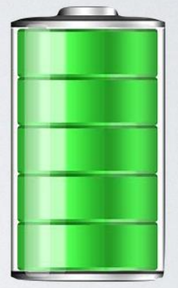
ENERGY SMELLS

470 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 44, NO. 5, MAY 2018

EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps

Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury

Abstract—Increased usage of mobile devices, such as smartphones and tablets, has led to widespread popularity and usage of mobile apps. If not carefully developed, such apps may demonstrate energy-inefficient behaviour, where one or more energy-intensive hardware components (such as Wifi, GPS, etc) are left in a high-power state, even when no apps are using these components. We refer to such kind of energy-inefficiencies as energy bugs. Executing an app with an energy bug causes the mobile device to exhibit poor energy consumption behaviour and a drastically shortened battery life. Since mobiles apps can have huge input domains, therefore exhaustive exploration is often impractical. We believe that there is a need for a framework that can systematically detect and fix energy bugs in mobile apps in a scalable fashion. To address this need, we have developed EnergyPatch, a framework that uses a combination of static and dynamic analysis techniques to detect, validate and repair energy bugs in Android apps. The use of a light-weight, static analysis technique enables EnergyPatch to quickly narrow down to the potential program paths along which energy bugs may occur. Subsequent exploration of these potentially buggy program paths using a dynamic analysis technique helps in validations of the reported bugs and to generate test cases. Finally, EnergyPatch generates repair expressions to fix the validated energy bugs. Evaluation with real-life apps from repositories such as F-droid and Github, shows that EnergyPatch is scalable and can produce results in reasonable amount of time. Additionally, we observed that the repair expressions generated by EnergyPatch could bring down the energy consumption on tested apps up to 60 percent.



INFRASTRUCTURE-AS-CODE SMELLS

Does Your Configuration Code Smell?

Tushar Sharma, Marios Fragkoulis and Diomidis Spinellis
Dept of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
{tushar,mfg,dds}@aueb.gr

ABSTRACT
Infrastructure as Code (IaC) is the practice of specifying computing system configurations through code, and managing them through traditional software engineering methods. The wide adoption of configuration management and increasing size and complexity of the associated code, prompt for assessing, maintaining, and improving the configuration code's quality. In this context, traditional software engineering knowledge and best practices associated with code quality management can be leveraged to assess and manage

1. INTRODUCTION
Infrastructure as Code (IaC) [13] is the practice of specifying computing system configurations through code, automating system deployment, and managing the system configurations through traditional software engineering methods. For example, a server farm that contains numerous nodes with different hardware configurations and different software package requirements can be specified using configuration management languages such as Puppet [39], Chef [37], CFEngine [4], or Ansible [1] and deployed automatically

SMELLS IN CONTINUOUS INTEGRATION PIPELINES

Use and Misuse of Continuous Integration Features
An Empirical Study of Projects that (mis)use Travis CI

Keheliya Gallaba, *Student Member, IEEE*, and Shane McIntosh, *Member, IEEE*

Automated Reporting of Anti-Patterns and Decay in Continuous Integration

Carmine Vassallo, Department of Informatics, University of Zurich, Zurich, Switzerland, vassallo@ifi.uzh.ch
Sebastian Proksch, Department of Informatics, University of Zurich, Zurich, Switzerland, proksch@ifi.uzh.ch
Harald C. Gall, Department of Informatics, University of Zurich, Zurich, Switzerland, gall@ifi.uzh.ch
Massimiliano Di Penta, Department of Engineering, University of Sannio, Benevento, Italy, dipenta@unisannio.it

COMMUNITY SMELLS

Beyond Technical Aspects: How Do Community Smells Influence the Intensity of Code Smells?

Fabio Palomba, *Member, IEEE*, Damian A. Tamburri, *Member, IEEE*, Francesca Arcelli Fontana, *Member, IEEE*, Rocco Oliveto, *Member, IEEE*, Andy Zaidman, *Member, IEEE*, Alexander Serebrenik, *Senior Member, IEEE*.

Abstract—Code smells are poor implementation choices applied by developers during software evolution that often lead to critical flaws or failure. Much in the same way, community smells reflect the presence of organizational and socio-technical issues within a software community that may lead to additional project costs. Recent empirical studies provide evidence that community smells are often—if not always—connected to circumstances such as code smells. In this paper we look deeper into this connection by conducting a mixed-methods empirical study of 117 releases from 9 open-source systems. The qualitative and quantitative sides of our mixed-methods study were run in parallel and assume a mutually-confirmative connotation. On the one hand, we survey 162 developers of the 9 considered systems to investigate whether developers perceive relationship between community smells and the code smells found in those projects. On the other hand, we perform a fine-grained analysis into the 117 releases of our dataset to measure the extent to which community smells impact code smell intensity (i.e., criticality). We then propose a code smell intensity prediction model that relies on both technical and community-related aspects. The results of both sides of our mixed-methods study lead to one conclusion: community-related factors contribute to the intensity of code smells. This conclusion supports the joint use of community and code smells detection as a mechanism for the joint management of technical and social problems around software development communities.

TEST SMELLS

An Empirical Analysis of the Distribution of Unit Test Smells and Their Impact on Software Maintenance

Gabriele Bavota[1], Abdallah Qusef[1], Rocco Oliveto[2], Andrea De Lucia[1], David Binkley[3]

An Empirical Investigation into the Nature of Test Smells

Michele Tufano[1], Fabio Palomba[2], Gabriele Bavota[3], Massimiliano Di Penta[4], Rocco Oliveto[5], Andrea De Lucia[2], Denys Poshyvanyk[1]

Affect maintainability…

…but developers rarely perceive them as important

LINGUISTIC ANTIPATTERNS

Linguistic antipatterns: what they are and how developers perceive them

Venera Arnaoudova · Massimiliano Di Penta · Giuliano Antoniol

Published online: 29 January 2015
© Springer Science+Business Media New York 2015

Abstract Antipatterns are known as poor solutions to recurring problems. For example, Brown et al. and Fowler define practices concerning poor design or implementation solutions. However, we know that the source code lexicon is part of the factors that affect the psychological complexity of a program, i.e., factors that make a program difficult to understand and maintain by humans. The aim of this work is to identify recurring poor practices related to inconsistencies among the naming, documentation, and implementation of an entity—called Linguistic Antipatterns (LAs)—that may impair program understanding. To

Perceived as serious concerns by developers

When present in APIs, correlate with the increase of Stack Overflow questions [Aghajani et al.]

SQL SMELLS

Tool for detecting query antipatterns from Bill Karwins SQL Antipatterns catalog

A Static Code Smell Detector for SQL Queries Embedded in Java Code

Csaba Nagy*, Anthony Cleve†
PReCISE Research Center, University of Namur, Belgium
*csaba.nagy@unamur.be, †anthony.cleve@unamur.be

Abstract—A database plays a central role in the architecture of an information system, and the way it stores the data delimits its main features. However, it is not just the data that matters. The way it is handled, i.e., how the application communicates with the database is of critical importance too. Therefore the implementation of such a communication layer has to be reliable and efficient. SQL is a popular language to query a database, and modern technologies rely on it (or its dialects) as query strings embedded in the application code. In many languages (e.g. in Java), an embedded query is typically constructed through several string operations that obstruct developers in

technologies to communicate with a DB, while JDBC occurs as the only database framework in 56.3% of the projects [3].

Database access technologies intend to help developers in various ways. They make it easier to integrate the communication with the database into the application code, e.g., by providing a link between Java classes and database entities (e.g. ORMs), or merely by supporting to reuse and construct queries (e.g. prepared statements). However, as a drawback, a developer hardly sees the final SQL query that is, in the

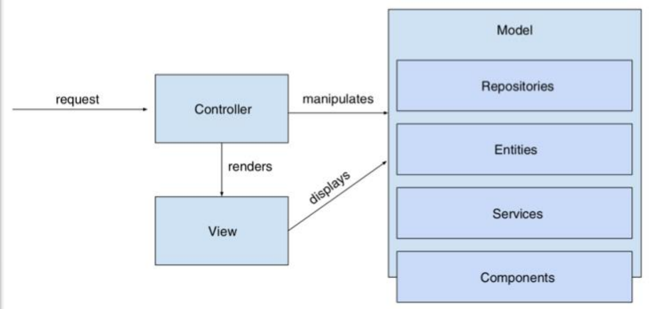
CODE SMELLS IN MODEL-VIEW CONTROLLER ARCHITECURES

Code smells for Model-View-Controller architectures

Maurício Aniche[1] · Gabriele Bavota[2] · Christoph Treude[3] · Marco Aurélio Gerosa[4] · Arie van Deursen[1]

Published online: 12 September 2017
© The Author(s) 2017. This article is an open access publication

Abstract Previous studies have shown the negative effects that low-quality code can have on maintainability proxies, such as code change- and defect-proneness. One of the symptoms of low-quality code are code smells, defined as sub-optimal implementation choices. While this definition is quite general and seems to suggest a wide spectrum of smells that can affect software systems, the research literature mostly focuses on the set of smells defined in



SERVICE ORIENTED ARCHITECTURE SMELLS

Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces

Mathieu Nayrolles, Naouel Moha, and Petko Valtchev

Investigating the Change-proneness of Service Patterns and Antipatterns

Francis Palma*†, Le An‡, Foutse Khomh‡, Naouel Moha† and Yann-Gaël Guéhéneuc*
*Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
†Latece, Département d'informatique, Université du Québec à Montréal, Canada
‡SWAT, DGIGL, École Polytechnique de Montréal, Canada
Email: {francis.palma, le.an, foutse.khomh, yann-gael.gueheneuc}@polymtl.ca, moha.naouel@uqam.ca

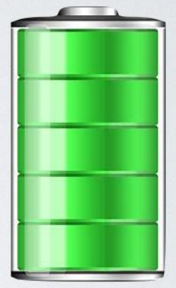
ENERGY SMELLS

470 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 44, NO. 5, MAY 2018

EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps

Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury

Abstract—Increased usage of mobile devices, such as smartphones and tablets, has led to widespread popularity and usage of mobile apps. If not carefully developed, such apps may demonstrate energy-inefficient behaviour, where one or more energy-intensive hardware components (such as Wifi, GPS, etc) are left in a high-power state, even when no apps are using these components. We refer to such kind of energy-inefficiencies as energy bugs. Executing an app with an energy bug causes the mobile device to exhibit poor energy consumption behaviour and a drastically shortened battery life. Since mobile apps can have huge input domains, therefore exhaustive exploration is often impractical. We believe that there is a need for a framework that can systematically detect and fix energy bugs in mobile apps in a scalable fashion. To address this need, we have developed EnergyPatch, a framework that uses a combination of static and dynamic analysis techniques to detect, validate and repair energy bugs in Android apps. The use of a light-weight, static analysis technique enables EnergyPatch to quickly narrow down to the potential program paths along which energy bugs may occur. Subsequent exploration of these potentially buggy program paths using a dynamic analysis technique helps in validations of the reported bugs and to generate test cases. Finally, EnergyPatch generates repair expressions to fix the validated energy bugs. Evaluation with real-life apps from repositories such as F-droid and Github, shows that EnergyPatch is scalable and can produce results in reasonable amount of time. Additionally, we observed that the repair expressions generated by EnergyPatch could bring down the energy consumption on tested apps up to 60 percent.



INFRASTRUCTURE-AS-CODE SMELLS

Does Your Configuration Code Smell?

Tushar Sharma, Marios Fragkoulis and Diomidis Spinellis
Dept of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
{tushar,mfg,dds}@aueb.gr

ABSTRACT

Infrastructure as Code (IaC) is the practice of specifying computing system configurations through code, and managing them through traditional software engineering methods. The wide adoption of configuration management and increasing size and complexity of the associated code, prompt for assessing, maintaining, and improving the configuration code's quality. In this context, traditional software engineering knowledge and best practices associated with code quality management can be leveraged to assess and manage

1. INTRODUCTION

Infrastructure as Code (IaC) [13] is the practice of specifying computing system configurations through code, automating system deployment, and managing the system configurations through traditional software engineering methods. For example, a server farm that contains numerous nodes with different hardware configurations and different software package requirements can be specified using configuration management languages such as Puppet [39], Chef [37], CFEngine [4], or Ansible [1] and deployed automatically

SMELLS IN CONTINUOUS INTEGRATION PIPELINES

Use and Misuse of Continuous Integration Features

An Empirical Study of Projects that (mis)use Travis CI

Keheliya Gallaba, *Student Member, IEEE,* and Shane McIntosh, *Member, IEEE*

Automated Reporting of Anti-Patterns and Decay in Continuous Integration

Carmine Vassallo	Sebastian Proksch	Harald C. Gall	Massimiliano Di Penta
Department of Informatics	*Department of Informatics*	*Department of Informatics*	*Department of Engineering*
University of Zurich	*University of Zurich*	*University of Zurich*	*University of Sannio*
Zurich, Switzerland	Zurich, Switzerland	Zurich, Switzerland	Benevento, Italy
vassallo@ifi.uzh.ch	proksch@ifi.uzh.ch	gall@ifi.uzh.ch	dipenta@unisannio.it

COMMUNITY SMELLS

Beyond Technical Aspects: How Do Community Smells Influence the Intensity of Code Smells?

Fabio Palomba, *Member, IEEE,* Damian A. Tamburri, *Member, IEEE,* Francesca Arcelli Fontana, *Member, IEEE,* Rocco Oliveto, *Member, IEEE,* Andy Zaidman, *Member, IEEE,* Alexander Serebrenik, *Senior Member, IEEE.*

Abstract—Code smells are poor implementation choices applied by developers during software evolution that often lead to critical flaws or failure. Much in the same way, community smells reflect the presence of organizational and socio-technical issues within a software community that may lead to additional project costs. Recent empirical studies provide evidence that community smells are often—if not always—connected to circumstances such as code smells. In this paper we look deeper into this connection by conducting a mixed-methods empirical study of 117 releases from 9 open-source systems. The qualitative and quantitative sides of our mixed-methods study were run in parallel and assume a mutually-confirmative connotation. On the one hand, we survey 162 developers of the 9 considered systems to investigate whether developers perceive relationship between community smells and the code smells found in those projects. On the other hand, we perform a fine-grained analysis into the 117 releases of our dataset to measure the extent to which community smells impact code smell intensity (i.e., criticality). We then propose a code smell intensity prediction model that relies on both technical and community-related aspects. The results of both sides of our mixed-methods study lead to one conclusion: community-related factors contribute to the intensity of code smells. This conclusion supports the joint use of community and code smells detection as a mechanism for the joint management of technical and social problems around software development communities.

ANTIPATTERNS AND SMELLS IN MULTI-LANGUAGE SYSTEMS

Forthcoming @EuroPLOP'19, CASCON'19, TPLOP

Anti-Patterns for Multi-language Systems

Mouna Abidi
Polytechnique Montreal
mouna.abidi@polymtl.ca

Manel Grichi
Polytechnique Montreal
manel.grichi@polymtl.ca

Yann-Gaël Guéhéneuc
Concordia University
yann-gael.gueheneuc@concordia.ca

Foutse Khomh
Polytechnique Montreal
foutse.khomh@polymtl.ca

ABSTRACT

Multi-language systems are common nowadays because most of the systems are developed using components written in different programming languages. These systems could arise from three different reasons: (1) to leverage the strengths and take benefits of each language, (2) to reduce the cost by reusing code written in other languages, (3) to include and accommodate legacy code.

Software quality is one of the most important cor to reduce testings, maintenance, and evolution c

Software quality partly depends on adopting g patterns, and avoiding code smells and design example, design patterns [4] describe good solu design problems. On the contrary, design anti- poor solutions to design problems [5, 6]. Design

Code Smells for Multi-language Systems

Mouna Abidi
Polytechnique Montreal
mouna.abidi@polymtl.ca

Manel Grichi
Polytechnique Montreal
manel.grichi@polymtl.ca

Yann-Gaël Guéhéneuc
Concordia University
yann-gael.gueheneuc@concordia.ca

Foutse Khomh
Polytechnique Montreal
foutse.khomh@polymtl.ca

ABSTRACT

Software quality becomes a necessity and no longer an advantage. In fact, with the advancement of technologies, companies must provide software with good quality. Many studies introduce the use of design patterns as improving software quality and discuss the

and HTML [1]. Most of the systems with which we interact daily are built using a combination of programming languages, such as Facebook, Youtube, etc[2]. Developers can reuse existing modules and components, without writing the source code from scratch [3]. They often choose the programming language most suitable for

TAKEAWAYS - SMELLS

Bad practices in (software) development are everywhere, beyond source code

Smell detectors to be used to trigger alarms and prevent future problems, rather than for predicting

TAKE AWAYS - GENERAL

The magnitude of a phenomenon might change based on the angle from which one observes it

Intensity more important than mere presence/absence of symptoms

CONCLUSION

Results (1/6)

Results RQ1 (Azureus)

Releases	Smells-Changes	Smells-No Changes	No Smells-Changes	No Smells-No Changes	p-values	OR
3.1.0.0	220	1967	20	1433	$< \mathbf{0.01}$	8.01
3.1.1.0	564	1686	101	1381	$< \mathbf{0.01}$	4.57
4.0.0.0	83	2238	7	1519	$< \mathbf{0.01}$	8.05
4.0.0.2	106	2206	12	1510	$< \mathbf{0.01}$	6.04
4.0.0.4	435	1886	39	1484	$< \mathbf{0.01}$	8.77
4.1.0.0	50	2297	11	1533	$< \mathbf{0.01}$	3.03
4.1.0.2	112	2235	11	1533	$< \mathbf{0.01}$	6.98
4.1.0.4	112	2236	12	1532	$< \mathbf{0.01}$	6.39
4.2.0.0	37	2353	3	1580	$< \mathbf{0.01}$	8.28

15/27

Results (4/6)

- Results RQ2 (Eclipse)

Releases	M-W p	t-test p	Cohen d
1.0	0.79	**0.03**	0.06
2.0	$< \mathbf{0.01}$	$< \mathbf{0.01}$	−0.08
2.1.1	$< \mathbf{0.01}$	$< \mathbf{0.01}$	0.31
2.1.2	$< \mathbf{0.01}$	$< \mathbf{0.01}$	0.13
2.1.3	**0.04**	$< \mathbf{0.01}$	0.07
3.0	0.07	0.10	0.03
3.0.1	0.11	0.26	−0.03
3.0.2	0.12	0.28	−0.02
3.2	$< \mathbf{0.01}$	$< \mathbf{0.01}$	0.41
3.2.1	$< \mathbf{0.01}$	$< \mathbf{0.01}$	0.29
3.2.2	$< \mathbf{0.01}$	$< \mathbf{0.01}$	0.25
3.3	$< \mathbf{0.01}$	$< \mathbf{0.01}$	0.41
3.3.1	$< \mathbf{0.01}$	$< \mathbf{0.01}$	0.18

18/27

Results (6/6)

- Results RQ3 (Eclipse)

Smells	Proneness to Changes
AbstractClass	1
ChildClass	6
ClassGlobalVariable	2
ClassOneMethod	4
ComplexClassOnly	8
ControllerClass	4
DataClass	4
FewMethods	2
FieldPrivate	6
FieldPublic	8
FunctionClass	1
HasChildren	**11**
LargeClass	8
LargeClassOnly	–
LongMethod	9
LongParameterListClass	6
LowCohesionOnly	5
ManyAttributes	9
MessageChainsClass	**10**
MethodNoParameter	8
MultipleInterface	5
NoInheritance	–
NoPolymorphism	3
NotAbstract	1
NotComplex	**10**
OneChildClass	2
ParentClassProvidesProtected	–
RareOverriding	4
TwoInheritance	–

20/27

Discussion

- Classes with smells are more change-prone, some odds ratio 3 to 8 times bigger for these classes.
- HasChildren, MessageChains, NotComplex, and NotAbstract lead almost consistently to change-prone classes.
- Existing smells are generally removed from the system while some new are introduced in the context of new features addition.

21/27

SMELLS LIKE TEEN SPIRIT…

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

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

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

prediction model can contribute to the correct classification of the buggyness of such a component. To verify this conjecture, we use the intensity index (*i.e.,* a metric able to estimate the severity of a code smell) defined by Arcelli Fontana *et al.* [31] to build a bug prediction model that takes into account the presence and the severity of design problems affecting a code component. Specifically, we evaluate the predictive power of the intensity index by adding it in a bug prediction model based on structural quality metrics [32], and comparing its accuracy against the one achieved by the baseline model on six large Java open source systems. We also quantified the gain provided by the addition of the intensity index with respect to the other structural metrics in the model, including the ones used to compute the intensity. Finally, we report further analyses aimed at understanding (i) the accuracy of a model

Smell intensity more important than other metrics for predicting fault-proneness

TEST SMELLS

An Empirical Analysis of the Distribution of Unit Test Smells and Their Impact on Software Maintenance

An Empirical Investigation into the Nature of Test Smells

Affect maintainability…

…but developers rarely perceive them as important

LINGUISTIC ANTIPATTERNS

Linguistic antipatterns: what they are and how developers perceive them

Venera Arnaoudova · Massimiliano Di Penta · Giuliano Antoniol

Perceived as serious concerns by developers

When present in APIs, correlate with the increase of Stack Overflow questions [Aghajani et al.]

SQL SMELLS

Tool for detecting query antipatterns from Bill Karwins SQL Antipatterns catalog

A Static Code Smell Detector for SQL Queries Embedded in Java Code

CODE SMELLS IN MODEL-VIEW CONTROLLER ARCHITECTURES

Code smells for Model-View-Controller architectures

Maurício Aniche · Gabriele Bavota · Christoph Treude · Marco Aurélio Gerosa · Arie van Deursen

SERVICE ORIENTED ARCHITECTURE SMELLS

Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces

Investigating the Change-proneness of Service Patterns and Antipatterns

ENERGY SMELLS

EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps

Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury

INFRASTRUCTURE-AS-CODE SMELLS

Does Your Configuration Code Smell?

Tushar Sharma, Marios Fragkoulis and Diomidis Spinellis
Dept of Management Science and Technology
Athens University of Economics and Business
Athens, Greece

SMELLS IN CONTINUOUS INTEGRATION PIPELINES

Use and Misuse of Continuous Integration Features

An Empirical Study of Projects that (mis)use Travis CI

Automated Reporting of Anti-Patterns and Decay in Continuous Integration

COMMUNITY SMELLS

Beyond Technical Aspects: How Do Community Smells Influence the Intensity of Code Smells?

TAKE AWAYS - GENERAL

The magnitude of a phenomenon might change based on the the angle from what you observe it

Intensity more important than mere presence/absence of symptoms