University of Salerno

CODE SMELLS: RELEVANCE OF THE PROBLEM AND NOVEL DETECTION TECHNIQUES

Fabio Palomba

A thesis submitted for the degree of
Doctor of Philosophy

December 2016

SOFTWARE ENGINEERING LAB, UNIVERSITY OF SALERNO

HTTP://WWW.SESA.UNISA.IT

This research was done under the supervision of Prof. Andrea De Lucia and Prof. Rocco Oliveto with the financial support of the University of Salerno and the University of Molise, from January 1st of 2014 to December 31th of 2016.

The final version of the thesis has been revised based on the reviews made by Prof. Michele Lanza and Prof. Jens Krinke.

The doctoral committee was composed by:

Prof. Dr. Filomena Ferrucci	Chairperson, University of Salerno, Italy
Prof. Dr. Giovanni Acampora	University of Naples "Federico II", Italy
Prof. Dr. Anna Rita Fasolino	University of Naples "Federico II", Italy
Prof. Dr. Michele Lanza	University of Lugano, Switzerland

First release, December 2016. Revised on March 2017.

Some Preliminary Considerations and Acknowledgments

I was lucky.

I started my PhD already counting a quite important ASE paper (about the following cited HIST) that would became a TSE soon. In that period, I had my first meeting as a new PhD Student with my boss, **Andrea De Lucia**. I expected honey words, but I was surprised when he told me: *"You did a good job, however the time of HIST is almost done. What's next? What do you want to do after HIST?"*. I was somehow shocked: I had no time to relax a bit after my good work, that I should have been already prepared for the next one. Only some months later, everything was clear. I got the real meaning of that words and I learned the lesson: success is a long and winding road, and the only way to grow up is to continue working hard. Above all, this is what I learned from Andrea. The second huge lesson is that *"the sum of two good papers does not result in an excellent paper. You need to be focused on your objective, and provides a solution for a single problem per time"*. With the years, I realized that this rule holds for all the important things of life: overacting just for the sake of showing how strong you are does not make sense without a well-defined goal. All in all, I believe that Andrea has been more than a "simple" advisor.

At the same time I had a co-advisor, **Rocco Oliveto**. I spent most of my first PhD year at the University of Molise (at home, after all). I remember that during my first month I was working and I said to Rocco that I was in a hurry because

of an upcoming deadline of an important conference, where I would have liked to submit a paper. That was a fault. I received a terrible scolding, because *"you must not reason in terms of conferences, you must not think about the number of papers in your CV! You must simply work to do something good"*. Today I can say that he was right, I was simply too impulsive and completely unprepared. Even if our relationship has been not always a bowl of cherries, I must thank Rocco because he balanced my ambitions with the awareness to have to keep always one's feet on the ground. This is one of the main output of my PhD: as I said several times, all the PhD students should have a Rocco in their team.

For several reasons, I must deserve a special thank to **Gabriele Bavota**. He held my hand when I was too young, and he learned me how to behave and how to be a good researcher.

I academically grew up following the lessons learned by observing and talking with the people mentioned above. Thanks for all the effort spent and the suggestions you gave to me. I hope I have not disappointed the expectations.

During these years, I also had the pleasure and the honor to work with some of the most important and prolific researchers of the world. Their contribution in my academic growth as well as in the research presented in this thesis cannot be described by words. For this reason, I must thank **Massimiliano Di Penta**, **Denys Poshyvanyk**, **Annibale** and **Sebastiano Panichella**, **Fausto Fasano**, **Andrian Marcus**, **Mario Linares Vasquez**, **Filomena Ferrucci**, **Gemma Catolino**, **Harald Gall**, **Adelina Ciurumelea**, **Carmine Vassallo**, **Giovanni Grano**, **Damian Andrew Tamburri**, **Elisabetta Di Nitto**, **Francesca Arcelli Fontana**, **Marco Zanoni**, **Mauricio Aniche**, **Luca Pascarella**, **Moritz Beller**, **Qianqian Zhu**, **Alberto Bacchelli**, and **Arie Van Deursen**.

I was lucky.

I had the non-trivial opportunity to work together with friends, achieving some of the top results of my PhD.

Thanks to **Michele Tufano**: as I always said, the best of us. We passed from academic projects to the top of the world with the ICSE'15 Distinguished Paper Award. It was amazing, and I will never forget it.

Thanks to **Dario Di Nucci**: I love thinking we brought (and we are bringing) some fresh air into areas that have not seen breakthroughs for some time. We touched the top several times, and I'm sure it will happen again.

Thanks to **Pasquale Salza**: In all honestly, I will always remember him for the night before the submission of the ICSE'17 paper, where he carefully proofread every single word of the paper at 2 a.m., after a full day of (desperate) work. Fortunately, that paper was accepted: I am still really hoping that the acceptance was due to that careful proofreading.

My PhD has been also composed of funny moments. Unfortunately, or perhaps fortunately, the SESA LAB has seen a lot of them. Anyway, I believe that the research group hosted in this laboratory has delivered more than the required and much more than the expected. I really hope to see this research group in the same laboratory again. For this reason, I *really* hope that the things will be easier in Italy, and in Salerno, soon.

As I said, I was lucky.

During my PhD, I had the pleasure to co-advise several Bachelor and Master Theses. Beyond doubts, students gave me more than they have received. For this reason, I want to thank **Antonio Panariello**, **Felice D'Avino**, **Stefano Ferrante**, **Marco Moretti**, **Emilio Arvonio**, **Gilda Trezza**, **Gianmarco Del Pozzo**, **Santolo Tubelli**, **Davide De Chiara**, **Alessandro Longo**, **Fabio Soggia**, **Giuseppe De Rosa**, **Sandro Siravo**, **Antonio Prota**, **Fabiano Pecorelli**, **Elena Sollai**, **Michele Lotierzo**, **Salvatore Geremia**, **Pasquale Martiniello**, **Dustin Lim**, and all the other students that had good (or even hard) times with me.

As I already said, I was lucky.

In a crisis period such as the one experienced by my country, and in particular, by my University, I had the opportunity to travel quite a lot. Thanks to the University of Salerno and the University of Molise for having supported me.

These trips gave me the opportunity to know and meet some important and somehow interesting characters of the research community. One of them is beyond doubts **Michele Lanza**. I knew him in Victoria during my first ICSME conference in 2014: on the Thursday before I left, he approached and said me important words that I hope he did not forget. After almost three years from that moment, he became one of the Reviewer of my PhD thesis. It has been a honor for me, and I hope that his final PhD Thesis review will always be in my mind during my future black periods (if any, and hopefully not).

Once again, I was lucky.

Up to now, I never sent a CV looking for a new position. This was also due to my visiting period at TU DELFT at the end of my second PhD year. During that period, I knew and had the honor to work together with **Andy Zaidman**. Afterwards, I don't know why and I don't know what he saw in me, but the fact is that he asked me to join TU DELFT for my post-doc: I accepted. I will be always grateful to Andy for the opportunity he gave to me. Besides being a (great) critique reviewer, and therefore a great advisor, Andy has also an enviable calm when facing every daily problem (even the more complex one). I *really* hope to get from him at least a 1% of his enormous qualities.

I was extremely lucky.

During my life as a PhD Student, I lived in a very small town close to the University. I always felt at home: who is not born around that town or that region cannot understand what I mean. One of the main reason for this statement (besides the food, of course) is the presence of an "additional family" who always waited for me after a workday (physically or not). **Alessandro** who cooked me some hot foods when I was late for dinner. **Luigi**, for having caused several delays because

of his long showers. **Cosmo** who became a (non-properly, but still valuable) masterchef for me. **Elisa**, who deserved to me some – sometimes right, sometimes wrong – scolding as well as some conversations abount life and future. **Dario**, for the philosophical conversations to keep always around a beer. **Pasquale**, for the amazing ideas which came after having drunk too much drinks (and also for the amazing time I spent seeing him after drunk too much drinks). **Ciro**, for the strength to carry on although the problems. **Enrico**, who will be always part of my heart. **Cristina**, who is my preferred (no matters that she is the one) "sister in law". **Felicia** and **Antonio** for having always treated me like a son. **Veronica**, for her amazing future and her future (amazing again) family. **Lucio**, who is always my family. **Gemma**, who is always Gemma. I will always thank and love you.

You now should know that I was really lucky. And I was even more lucky because of my "blood tie" family.

Thanks to **my father**, for carefully tracing every trip I do. Thanks **to my mother**, for saying "hello" at most every 5 hours, even if I am almost 30 years old. Thanks to **Gabriella**, for being a silent, but always noisy, part of my life. Thanks to **Michele**, who became part of our family before becoming a great father. And, much more importantly, thanks to **Elisa** for being the most beautiful (and smart) thing I saw in my life. Thanks to all for being always close to me, even though the distance.

Thanks to **Gemma**, who became part of my life exactly when I started my PhD, and who still does not desist from being part of my daily life. She made (and still makes) me a better person, under all the perspectives. I would not be the same without her, and for this reason I hope she will be part of my life for a long while. I love you.

Last, but (absolutely) not the least I would like to thank the people who cannot celebrate this success with me, but that I am sure would have been proud of me. I will never forget their lessons and their love.

I was lucky, and I want to repeat it since for several years I have refused to believe that luck has a strong impact on daily life. But today I must acknowledge that without the

people I met I could never become what I became (Lucio will never let me forget it). Thanks again to all of you.

The end of a story is always a mix of happiness and sadness. I hope to have met the expectations and created a bit of happiness in the people who believed in me. For sure, I am quite satisfied from what I did. However, there is still a long way to go. And I am already thinking to what's next.

April 20, 2017
Fabio

"Be my mirror, my soul, my shield,
my missionaries in a foreign field"
C. Martin - 2008.

Abstract

Software systems are becoming the core of the business of several industrial companies and, for this reason, they are getting bigger and more complex. Furthermore, they are subject of frantic modifications every day with regard to the implementation of new features or for bug fixing activities. In this context, often developers have not the possibility to design and implement ideal solutions, leading to the introduction of technical debt, *i.e.*, "not quite right code which we postpone making it right".

One noticeable symptom of technical debt is represented by the bad code smells, which were defined by Fowler to indicate sub-optimal design choices applied in the source code by developers. In the recent past, several studies have demonstrated the negative impact of code smells on the maintainability of the source code, as well as on the ability of developers to comprehend a software system. This is the reason why several automatic techniques and tools aimed at discovering portions of code affected by design flaws have been devised. Most of them rely on the analysis of the structural properties (*e.g.*, method calls) mined from the source code.

Despite the effort spent by the research community in recent years, there are still limitations that threat the industrial applicability of tools for detecting code smells. Specifically, there is a lack of evicence regarding (i) the circustamces leading to code smell introduction, (ii) the real impact of code smells on maintainability, since previous studies focused the attention on a limited number of software projects. Moreover, existing code smell detectors might be inadeguate for the detection of many code smells defined in literature. For instance, a number

of code smells are intrinsically characterized by how code elements change over time, rather than by structural properties extractable from the source code.

In the context of this thesis we face these specific challenges, by proposing a number of large-scale empirical investigations aimed at understanding (i) when and why smells are actually introduced, (ii) what is their longevity and the way developers remove them in practice, (iii) what is the impact of code smells on change- and fault-proneness, and (iv) how developers perceive code smells. At the same time, we devise two novel approaches for code smell detection that rely on alternative sources of information, *i.e.*, historical and textual, and we evaluate and compare their ability in detecting code smells with respect to other existing baseline approaches solely relying structural analysis.

The findings reported in this thesis somehow contradicts common expectations. In the first place, we demonstrate that code smells are usually introduced during the first commit on the repository involving a source file, and therefore they are not the result of frequent modifications during the history of source code. More importantly, almost 80% of the smells survive during the evolution, and the number of refactoring operations performed on them is dramatically low. Of these, only a small percentage actually removed a code smell. At the same time, we also found that code smells have a negative impact on maintainability, and in particular on both change- and fault-proneness of classes. In the second place, we demonstrate that developers can correctly perceive only a subset of code smells characterized by long or complex code, while the perception of other smells depend on the intensity with which they manifest themselves.

Furthermore, we also demonstrate the usefulness of historical and textual analysis as a way to improve existing detectors using orthogonal informations. The usage of these alternative sources of information help developers in correctly diagnose design problems and, therefore, they should be actively exploited in future research in the field.

Finally, we provide a set of open issues that need to be addressed by the research community in the future, as well as an overview of further future applications of code smells in other software engineering field.

Contents

Part 1

Introduction and Background

Chapter 1

Problem Statement

1.1 Context and Motivation

The success of every industrial company depends on a number of factors. Some of them are related to the way current business is managed, some others on how the future business is created. In the digital age, *innovation* is one of the key factors contributing to the success of any company and, in this scenario, *information systems* have a prominent role.

Software systems drastically changed the lives of both individuals and complex organizations through the definition of ever more sophisticated ways to create business and involve clients. Social networking, digital communication and cloud computing are just few examples of the new opportunities which came through the wave of information technology.

As a direct consequence, software systems are getting always bigger, complex and difficult to manage. Simultaneously, with the high diffusion of laptops and smartphones there is a growing need for software that meet also performance, reliability, and maintainability requirements, other than costs [1]. The result is that not only the development of software systems is more complex than before, but more importantly perfective and corrective maintenance operations need to be performend according to strict scheduling and deadlines. Indeed, it is no accident that maintenance costs are up to 100 times higher than development costs [2].

In a typical scenario, the source code of a software system experience several daily changes to be adapted to new contexts or to be fixed with regard to urgent bugs [3]. Due to unavoidable time constraints, software developers do not always have the time to design and implement correct solutions and, indeed, often the activities are performed in a rush, leading to the application of undisciplined implementation choices that have the effect to erode the original design of the system, possibly introducing what Cunningham defined as *technical debt* [4], *i.e.*, immature design solutions applied with the aim of meeting a deadline.

This phenomenon is widely known as *software aging* [5]. Previous research measured software aging exploiting change entropy [6, 7], while others focused the attention on the so-called *bad code smells*, recurring poor design or implementation choices applied as a consequence of software aging or because the source code of a system is not well-designed from the beginning [8].

For instance, a *Long Method* represents a method that implements a main functionality together with auxiliary functions that should be managed in different methods. Such methods can rapidly grow out of control, making it harder and harder for developers to understand them, to fix bugs or to add new features [8]. Together with long or complex classes (*e.g.*, Blob), poorly modularized code (*e.g.*, Promiscuous Package), or long *Message Chain*s used to develop a given feature, they are just few examples of code smells that can possibly negatively affect a software system [8].

In the last decade, code smells have attracted the attention of several researchers, interested in understanding the intrinsic characteristics of source code affected by design flaws as well as the effect of the presence of code smells on maintainability. Specifically, a plethora of empirical studies have investigated the relevance of code smells from the developers' perspective [9, 10, 11], their evolution and longevity [12, 13, 14, 15], and their side effects, such as the increase of change- and fault-proneness [16, 17], the decrease of program comprehensibility [18] and maintainability [9, 19, 10].

All these studies motivated the research community in investing effort for the definition of techniques able to automatically detect code smells in the source code,

with the aim of supporting developers in the identification of the portions of the source code more likely to be affected by design problems.

Indeed, several code smell detectors have been devised. Most of them proposed to detect code smells by (i) identifying key *symptoms* that characterize particular bad smells using a set of thresholds based on the measurement of structural metrics (*e.g.,* if Lines Of Code $\geqslant k$); (ii) conflating the identified symptoms, leading to the final rule for detecting the smells [20, 21, 22, 23, 24, 25, 26]. These detection techniques mainly differ in the set of used structural metrics, which depend on the type of code smells to detect, and how the identified key *symptoms* are combined. For example, such a combination can be performed using AND/OR operators [21, 26, 22], Bayesian belief networks [23], and B-Splines [24].

An alternative way to find code smells is to look for portions of code that need to be refactored. As an example, Tsantalis *et al.* devised *JDeodorant* [25], a tool able to detect instances of the *Feature Envy* smell by analyzing the structural dependencies of classes in order to suggest where to apply operations of *Move Method* refactorings. Following a similar phylosophy, Bavota *et al.* [27, 28] proposed approaches for recommending refactoring operations exploiting graph-theory [27, 29], game theory [28], and relational topic modeling [30, 31].

Finally, the usage of other methodologies such as search algorithms [32, 33, 34, 35] or machine learning [36, 37], as well as the definition of context-based detection techniques [38, 39] represent a recent trend in the field.

1.2 Research Statement

Even though the effort devoted by the research community in characterizing code smells through the conduction of empirical studies and devising approaches for their detection exhibiting good detection performances, in the context of our research we highlighted some aspects that might be improved, summarizable as follow:

1. **Previous research is mainly based on common wisdom rather than guided**

by the state-of-the-practice. Indeed, the smell detectors proposed in literature do not consider the circumstances that could have caused the smell introduction, thus providing a detection only based on structural properties characterizing the current version of a system. The misalignment between theory and practice can cause an additional problem: smells detectable by using current approaches might be far from what developers actually perceive as design flaws, thereby leading developers to not refactor smells [40]. In our opinion, to better support developers in planning actions to improve source code quality, it is imperative to have a contextualized understanding of the circumstances under which particular smells occur.

2. **Small-scale empirical investigations threat the generalizability of previous findings.** Most of the empirical studies assessing the impact of code smells on non-functional attributes of source code have been carried out on a small set of software systems. Thus, it is unclear the magnitude of such effects and whether previous findings can be generalizable. For instance, several studies investigated the effects of smells on the fault-proneness of classes. Besides Khomh *et al.* [17], also D'Ambros *et al.* [41], Li and Shatnawi [42], Olbrich *et al.* [43], and Gatrell and Counsell [44] found relationships between the presence of design flaws and faults. However, all these studies analyzed at most seven systems, that can have specific characteristics and, therefore, might be not representative.

3. **The proposed techniques still might not be adequate for detecting many of the smells described in literature.** In particular, while there are smells where the use of structural analysis is suitable (*e.g.,* the *Complex Class* smell, a class having a high cyclomatic complexity), there are also several other design flaws not characterized by structurally visible problems. For example, a *Divergent Change* occurs when a class is changed in different ways for different reasons [8]. In this case, the smell is intrinsically characterized by how source code changes over time and, therefore, the use of historical information may help in its identification. At the same time, existing approaches do

not take into account the vocabulary of the source code, that can be more effective in the identification of poorly cohesive or more complex classes, as already pointed out in previous research related to other software engineering tasks [45, 46]. For instance, a *Long Method* might be effectively identified by analyzing the textual scattering in the source code of a method.

The work presented in this thesis has the goal to overcome the limitations mentioned above, by performing large-scale empirical investigations and defining new code smell detectors complementary to the ones previously proposed. Specifically, the high-level research questions considered in this thesis are the following:

- **RQ1**: *When and why code smells are actually introduced by developers?*
- **RQ2**: *Are previous findings on the relationship between code smells and maintainability generalizable?*
- **RQ3**: *Are code smells actually perceived by developers as actual design problems?*
- **RQ4**: *How do approaches based on alternative sources of information, such as the historical and the textual one, perform in detecting code smells?*
- **RQ5**: *Are code smells detectable using alternative sources of information closer to developers' perception of design problems?*

The final goal is to provide developers with more usable detectors, able to (i) accurately detect design flaws taking into account the way smells are generally introduced in the source code, and (ii) propose recommendations that are closer to the developers perspective.

1.3 Research Contribution

This thesis provides several contributions aimed at answering the research questions formulated in the previous Section. Basically, the contribution of our work can be divided in two main groups. Table 1.1 reports, for each group, the list of aspects treated and the references to papers published or submitted. Specifically:

Table 1.1: Summary of the Thesis Contribution

Field	Aspect	Ref.
Empirical Studies	Circumstances behind Smell Introduction	[47]
	Code Smell Survivability and Causes of their Removal	[47]
	Impact of Code Smells on Source Code Maintainability	[48]
	Relationships Between Code Smells and Refactoring	[49]
	Developers' Perception of Code Smells	[50]
Novel Approaches	Using Change History Information for Detecting Code Smells	[51, 52]
	Using Information Retrieval Techniques for Detecting Code Smells	[53, 54]
	Developers' Perception of Historical Smells	[52]
	Developers' Evolution and Perception of Textual Smells	[54]

1. **Empirical Studies on Code Smells.** This category of studies aims at answering the first three research questions:

   - To answer **RQ1**, we conducted a mining software repository study aimed at understanding how code smells are introduced in practice. The study refers to publication [47], and focused the attention of when developers introduce code smells and under which circumstances they apply modifications having the effect to introduce smells.
   - To answer **RQ2**, the contributions are related to publications [47], [48], and [49]. In particular, we conducted (i) an investigation aimed at understanding the lifespan of code smells and the likely causes behind their removal, and (ii) a mining study investigating the impact of code smells on source code change- and fault-proneness, and (iii) an experimental investigation into the relationship between code smells and refactoring operations performed by developers.
   - To answer **RQ3**, we performed a qualitative study with developers, where we collected their opinions in order to investigate whether developers perceive code smells as actual design flaws. This contribution is presented in publication [50].

The findings achieved from this group of empirical studies (i) shed lights on

the actual practices followed by developers when introducting and removing code smells, (ii) reveal the real impact of design smells on the maintainability of source code, and (iii) helped in focusing the attention of the contruction of code smells detectors taking into account the design flaws actually perceived by developers.

2. **Novel Approaches for Code Smell Detection.** Based on the results achieved from the previous empirical studies and having observed that several code smells cannot easily identified by solely exploiting structural analysis, we answer **RQ4** and **RQ5** devising two novel approaches for code smell detection based on the analysis of historical and textual information. In particular:

   - We exploited the change history available in software repositories to extract the fine-grained code changes experienced by a software system. Then, we applied heuritics based on change frequency or co-change analysis to detect code smells intrinsically characterized by how code changes over time. Moreover, we conducted an empirical evaluations aimed at measuring the performances of the approach when compared to existing structural-based techniques. This contribution refers to publications [51] and [52].
   - Secondly, we employed our historical detector in a second empirical study aimed at verifying whether developers are able to perceive historical smells easier than structural ones. The contribution is presented in publication [52].
   - We defined a code smell detector based on Information Retrieval techniques [55] to detect code smells characterized by promiscuous responsabilities (*e.g.,* Blob). The technique has been evaluated in terms of its accuracy in detecting code smells when compared with state-of-the-art structural detectors. Publication [53] discusses this contribution.
   - In the second place, we conducted a software repository mining study aimed at evaluating how developers act differently on textually versus

structurally detected smell instances. We complemented our analysis by surveying industrial developers and software quality experts in order to understand what is the difference in the way they perceive and identify code smells detected in the two different manners. Publications [53] and [54] include these contributions.

The output of this piece of research consists of a new set of techniques able to effectively complement previous code smell detectors defined in literature.

Besides the contributions described above, two further common contributions were made:

- **Large-scale Empirical Studies.** All the studies conducted in our research have been conducted on a large set of software systems, in order to ensure the generalizabily of the findings.

- **Publicly Available Tools and Replication Packages.** The construction of several tools, scripts, and dataset was needed to effectively perform the analyses. We made all of them publicly available by providing tools (publications [56, 57, 58, 59]) and online replication packages. Moreover, we built a web application containing an open dataset of manually evaluated code smells, which is discussed in publication [60].

1.4 Structure of the Thesis

Before describing in depth the single contributions of this thesis, Chapter 2 overviews the related literature on code smells. The core of the thesis is organized in two main blocks, one for each group reported in Table 1.1. Each part consists of the chapters further describing the single contributions provided. Specifically:

Part I groups together the empirical studies aimed at answering the first three research questions:

- *When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away).* Chapter 3 discusses the empirical study where we analyzed when and why code smells are actually introduced in practice, what is their longevity, and what are the practices used by developers to remove code smells.

- *On the Diffuseness and the Impact on Maintainability of Code Smells: A Large Scale Empirical Investigation.* Chapter 4 reports the design and the results of the experiment aimed at analysis the impact of code smells on change- and fault-proneness.

- *Do They Really Smell Bad? A Study on Developers Perception of Bad Code Smells.* The study on the perception that developers have about code smells is reported in Chapter 5.

- *An Experimental Investigation on the Innate Relationship between Quality and Refactoring.* Chapter 6 describes an empirical study where we evaluated whether software quality (including code smells) drive the application of refactoring operations.

Part II reports the chapters related to the novel approaches devised:

- *Mining Version Histories for Detecting Code Smells.* Chapter 7 describes HIST, the change history-based approach defined to detect five types of code smells from the catalogue by Fowler [8]. It also reports the study aimed at evaluating the performances of the novel detector and its comparison with extisting structural-based detectors.

- *The Perception of Historical and Structural Smells.* Chapter 8 describes the empirical study aimed at evaluating the perception of historical smells from the developers' point of view.

- *A Textual-based Approach for Code Smell Detection.* TACO, the approach relying on Information Retrieval techniques for detecting five types of smells from the catalogue by Fowler [8] and Brown *et al.* [61] is presented in Chapter

9. The Chapter also discusses the study where we evaluated its accuracy and compared the performances with existing tools relying on structural analysis.

- *The Scent of a Smell: an Extensive Comparison between Textual and Structural Smells.* Chapter 10 discusses of the empirical studies conducted to evaluate (i) how developers act on textual and structural code smells, and (ii) how developers perceived design flaws detected with different approaches.

Finally, Chapter 11 concludes the thesis and discusses the future directions and challenges in code smell research.

Chapter 2

Related Work

In the past years code smells have been attracted the attention of several researchers, who were interested in devising automatic approaches for detecting them in the source code, as well as understanding their nature and their impact on non functional attributes of source code. In this chapter a comprehensive literature review about (i) methods and tools able to detect code smells in the source code, and (ii) empirical studies conducted to analyze the evolution and the impact of code smells on maintainability is reported. Moreover, we also report an overview of the empirical studies conducted in the closely related field of refactoring.

2.1 Literature Review on Automatic Code Smell Detection Techniques

Several techniques have been proposed in the literature to detect code smell instances affecting code components, and all of these take their cue from the suggestions provided by four well-known books: [8, 61, 62, 63]. The first one, by Webster [62] defines common pitfalls in Object Oriented Development, going from the project management down to the implementation. Riel [63] describes more than 60 guidelines to rate the integrity of a software design. The third one, by Fowler [8], describes 22 code smells describing for each of them the refactoring actions to take. Finally, Brown *et al.* [61] define 40 code antipatterns of different nature (*i.e.*,

architectural, managerial, and in source code), together with heuristics to detect them. Among all the code smells defined in the books mentioned above, only for a subset of them automatic approaches and tools have been proposed. Specifically, the research community mainly focused its attention on the smells characterizing long or complex code, as well as smells shown as harmful from maintainability. On the other hand, a lower effort has been devoted in most of the smells characterizing problems in the way developers use Object Oriented programming. In the following, we present the definition of the code smells studied in literature, as well as the approaches proposed for their identification.

2.1.1 Detecting Complex and Long Code Elements

Complex and long code elements represent a major issue for the maintainability of a software system, since it threats the ability of developers to comprehend and modify the source code [18, 16, 17]. Moreover, such smells are generally perceived as actual problems from developers, as shown later in Chapter 5. For this reason, the research community devised approaches for automatic detection of a variety of code smells in the catalogues by Fowler [8] and Brown *et al.* [61].

The God Class. The *God Class* (*a.k.a., Blob*) is a class implementing different responsibilities, characterized by the presence of a large number of attributes and methods, which implement different functionalities, and by many dependencies with data classes (*i.e.,* classes implementing only getter and setter methods) [8]. A suitable way to remove this smell is characterized by the application of an Extract Class Refactoring operation aimed at splitting the original class in more cohesive classes [8].

The problem to identify classes affected by the *Blob* code smell has been analyzed under three perspectives. First, researchers focused their attention on the definition of heuristic-based approaches that exploit several software quality metrics (*e.g.,* cohesion [64]). For instance, Travassos *et al.* [65] defined the "reading techniques", a mechanism suggesting manual inspection rules to identify defects in source code.

```
RULE_CARD: Blob {
  RULE: Blob
        {ASSOC: associated FROM: mainClass ONE TO: DataClass MANY}
  RULE: mainClass
        {UNION LargeClassLowCohesion ControllerClass}
  RULE: LargeClassLowCohesion
        {UNION LargeClass LowCohesion}
  RULE: LargeClass
        {(METRIC: NMD + NAD, VERY_HIGH, 20)}
  RULE: LowCohesion
      {(METRIC: LCOM5, VERY_HIGH, 20)}
  RULE: ControllerClass
      {UNION (SEMANTIC: METHODNAME, {Process, Control, Command, Manage, Drive, System}),
         (SEMANTIC: CLASSNAME, {Process, Control, Command, Manage, Drive, System}}
  RULE: DataClass
      {(STRUCT: METHOD_ACCESSOR, 90)} };
```

Listing 2.1: Rule Card used by DECOR for detecting Blob instances.

DECOR (DEtection and CORrection of Design Flaws) [20], use a set of rules, called "rule card", describing the intrinsic characteristics of a class affected by Blob (see Listing 2.1). As described in the rule card, DECOR detects a Blob when the class has an LCOM5 (Lack of Cohesion Of Methods) [64] higher than 20, a number of methods and attributes higher than 20, a name that contains a suffix in the set {Process, Control, Command, Manage, Drive, System}, and it has a one-to-many association with data classes.

Besides DECOR, Marinescu [21] proposed a metric-based mechanism to capture deviations from good design principles and heuristics, called "detection strategies". Such strategies are based on the identification of *symptoms* characterizing a particular smell and *metrics* for measuring such symptoms. Then, thresholds on these metrics are defined in order to define the rules. Lanza and Marinescu [22] showed how to exploit quality metrics to identify "disharmony patterns" in code by defining a set of thresholds based on the measurement of the exploited metrics in real software systems. Their detection strategies are formulated in four steps. In the first step, the *symptoms* characterizing a smell are defined. In the second step, a proper *set of metrics* measuring these symptoms is identified. Having this information, the next step is to define thresholds to classify the class as affected (or not) by the defined symptoms. Finally, AND/OR operators are used to correlate the symptoms, leading to the final rules for detecting the smells.

The approaches described above classify classes strictly as being clean or anti-

patterns, while an accurate analysis for the borderline classes is missing [23]. In order to bridge this gap, Khomh *et al.* [23] proposed an approach based on Bayesian belief networks providing a likelihood that a code component is affected by a smell, instead of a boolean value as done by the previous techniques. This is also one of the main characteristics of the approach based on the quality metrics and B-splines proposed by Oliveto *et al.* [24] for identifying instances of *Blobs* in source code. Given a set of Blob classes it is possible to derive their signature (represented by a curve) that synthetize the quality of the class. Specifically, each point of the curve is the value of a specific quality metrics (*e.g.,* the CK metric suite). Then, the identification of Blob is simply obtained by comparing the curve (signature) of a class given in input with the (curves) signatures of the previous identified Blob instances. The higher the similarity, the higher the likelihood that the new class is a Blob as well.

Blob classes might be also detected indirectly by looking at the opportunity to apply an Extract Class Refactoring. For instance, Fokaefs *et al.* [66] proposed an approach that takes as input a software system and suggests a set of Extract Class Refactoring operations. In other words, the tool suggests to split a set of classes in several classes in order to have a better distribution of the responsibilities. Clearly, the original classes are candidate Blob instances. The approach proposed by Fokaefs *et al.* [66] formulated the detection of Extract Class Refactoring operations as a cluster analysis problem, where it is necessary to identify the optimal partitioning of methods in different classes. In particular, for each class they analyze the structural dependencies between the entities of a class, *i.e.,* attributes and methods, in order to build, for each entity, the entity set, *i.e.,* the set of methods using it. The Jaccard distance between all couples of entity sets of the class is then computed in order to cluster together cohesive groups of entities that can be extracted as separate classes. The Jaccard distance is computed as follows:

$$\text{Jaccard}(E_i, E_j) = 1 - \frac{|E_i \cap E_j|}{|E_i \cup E_j|} \tag{2.1}$$

where E_i and E_j are two entity sets, the numerator is the number of common

entities between the two sets and the denominator is the total number of unique entities in the two sets. If the overall quality (in terms of cohesion and coupling) of the system improves after splitting the class in separate classes, the approach proposed the splitting as a candidate Extract Class Refactoring operation. In other words, a Blob has been identified. The approach proposed by Fokaefs *et al.* [66] has been implemented as an Eclipse plug-in, called *JDeodorant* [67].

Similarly to this work, Bavota *et al.* [27] proposed the use of structural and conceptual analysis to support the identification of Extract Class Refactoring opportunities. In particular, a class of the system under analysis is first parsed to build a *method-by-method* matrix, a $n \times n$ matrix where *n* is the number of methods in the class to be refactored. A generic entry $c_{i,j}$ of the *method-by-method* matrix represents the likelihood that method m_i and method m_j should be in the same class. This likelihood is computed as a hybrid coupling measure between methods (degree to which they are related) obtained through a weighted average of three structural and semantic measures, *i.e.*, the Structural Similarity between Methods (SSM) [68], the Call-based Dependence between Methods (CDM) [69], and the Conceptual Similarity between Methods (CSM) [70]. Once the *method-by-method* matrix has been constructed, its transitive closure is computed in order to extract chains of strongly related methods (each chain represents the set of responsibilities, *i.e.*, methods, that should be grouped in a new class).

Bavota *et al.* [28] also devised the use of game theory to find a balance between class cohesion and coupling when splitting a class with different responsibilities into several classes. Specifically, the sequence of refactoring operations is computed using a refactoring game, in which the Nash equilibrium [71] defines the compromise between coupling and cohesion.

Finally, Simon *et al.* [72] provided a metric-based visualization tool able to discover design defects representing refactoring opportunities. For example, a *Blob* is detected if different sets of cohesive attributes and methods are present inside a class. In other words, a *Blob* is identified when there is the possibility to apply an Extract Class refactoring.

The Long Method. As defined by Fowler, a *Long Method* arises when a method implements a main functionality, together with auxiliary functions that should be managed in different methods [8]. Generally, this smell has been captured by considering the size (*i.e.,* lines of code) of a method. Indeed, Moha *et al.* [20] identified the smell using a simple heuristic based on the LOC metric. Indeed, if a method contains more than 100 lines of code, a Long Method is detected. A more sophisticated approach has been proposed by Tsantalis and Chatzigeorgiou [73] for detecting Extract Method Refactoring opportunities. Specifically, the technique employs a block-based slicing technique [74] in order to suggest slice extraction refactorings which contain the complete computation of a given variable. If it is possible to extract a slice for a parameter, an Extract Method Refactoring can be applied. Consequentely, a Long Method is identified.

```
RULE_CARD: Spaghetti Code {
  RULE: Spaghetti Code
        {{INTER: NoInheritanceClassGlobalVariable LongMethodMethodNoParameter}
  RULE: LongMethodMethodNoParameter
        {INTER LongMethod MethodNoParameter}
  RULE: LongMethod
        {(METRIC: METHOD_LOC, VERY_HIGH, 100)}
  RULE: MethodNoParameter
        {(STRUCT: METHOD_NO_PARAM)}
  RULE: NoInheritanceClassGlobalVariable
      {INTER NoInheritance ClassGlobalVariable}
  RULE: NoInheritance
      {(METRIC: DIT, INF_EQ, 2, 0)}
  RULE: ClassGlobalVariable
      {INTER ClassOneMethod FieldPrivate}
  RULE: ClassOneMethod
      {(STRUCT: ONE_METHOD)}
  RULE: FieldPrivate
      {(STRUCT: PRIVATE_FIELD, 100)};
```

Listing 2.2: Rule Card used by DECOR for detecting Spaghetti Code instances.

The Spaghetti Code. Classes affected by this smell are characterized by complex methods with no parameters, interacting between them using instance variables. It is a symptom of procedural-style programming [61]. The *Spaghetti Code* smell describes source code difficult to comprehend by a developer, often without a well defined structure, defining several long methods without any parameter. From a lexical point of view, smelly classes have usually procedural names. DECOR [20] is also able to identify Spaghetti Code, once again by using a specific rule card that describes the smell through structural properties.

As the reader can see in Listing 2.2, DECOR classifies the smell using only software metrics able to identify the specific characteristic of a class. This is possible simply because the Spaghetti Code does not involve any relationships with other classes, but it is a design problem concentrated in a single class having procedural characteristics. Specifically, in DECOR instances of Spaghetti Code are found looking for classes having at least one long method, namely a method composed by a large number of LOC and declaring no parameters. At the same time, the class does not present characteristics of Object Oriented design. For example, the class does not use the concept of inheritance and should use many global variables.

The Swiss Army Knife. A *Swiss Army Knife* is a class that exhibits high complexity and offers a large number of different services. This type of smell is slightly different from a Blob, because in order to address the different responsibilities, it exposes high complexity while a Blob is a class that monopolizes processing and data of the system.

In other words, a class affected by the Swiss Army Knife is a class that provides answer to a large range of needs. Generally, it arises when a class has many methods with high complexity and the class has a high number of interfaces. For example, a utility class exposing high complexity and addressing many services is a good candidate to be a Swiss Army Knife. The characteristics of this design flaw suggest that structural information can be useful for its detection. In particular, a mix of software complexity metrics and semantic checks in order to identify the different services provided by the class could be used for the detection. Once again, DECOR is able to detect this smell through a rule card [20]. Specifically, it relies on the Number of Interfaces metric, which is able to identify the number of services provided by a class. If the metric exceeds a given threshold, a Swiss Army Knife is detected.

The Type Checking. A *Type Checking* code smell affects a class that shows complicated conditional statements making the code difficult to understand and maintain. One of the most common situation in which a programmer linked to procedural languages can fall down using Object Oriented programming is the misun-

derstanding or lack of knowledge on how to use OO mechanisms such as polymorphism. This problem manifests itself especially when a developer uses conditional statements to dynamically dispatch the behavior of the system instead of polymorphism.

For its identification, a simple heuristic can be used: when you see long and intricate conditional statements, then you have found a Type Checking candidate. Tsantalis *et al.* [75] proposed a technique to identify and refactor instances of this smell. In particular, their detection strategy takes into account two different cases: in the first case, an attribute of a class represents a state and, depending on its value, different branches of the conditional statements are executed. Thus, if in the analyzed class, there is more than one condition involving the attribute, a candidate of Type Checking is found. In the second case, a conditional statement involves RunTime Type Identification (RTTI) in order to cast the type of a class in another to invoke methods on the last one. For example, this is the case of two classes involved in a hierarchy where the relationship is not exploited by using polymorphism. In such case, RTTI often arises in the form of an if/else if statement in which there is one conditional expression for each control on the type of a class. However, if the RTTI involves more than one conditional expression, a candidate of Type Checking is found.

2.1.2 Detecting Wrong Application of Object Oriented Programming

Even though not immediately visible, wrong application of Object Oriented programming can lead to a higher change- and fault-proneness of the source code [16, 17, 41]. Most of the approaches defined in literature focus their attention on four particular smell types, *i.e.*, *Feature Envy*, *Refused Bequest*, and *Functional Decomposition*.

The Feature Envy. A method suffers of the *Feature Envy* code smell if it is more interested in another class (also named *envied class* with respect the one it actually is in). It is often characterized by a large number of dependencies with the envied

class [8]. Usually, this negatively influences the cohesion and the coupling of the class in which the method is implemented. In fact, the method suffering of Feature Envy reduces the cohesion of the class because it likely implements different responsibilities with respect to those implemented by the other methods of the class and increases the coupling, due to the many dependencies with methods of the envied class.

A first simple way to detect this smell in source code is to traverse the Abstract Syntax Tree (AST) of a software system in order to identify, for each field, the set of the referencing classes [76]. So, using a threshold it is possible to discriminate the fields having too many references with other classes. Other more sophisticated techniques defined in literature are able to identify *Move Method* Refactoring, *i.e.*, operations aimed at removing the Feature Envy smell [25, 30]. In some way, these approaches can aid the software engineer also in the identification of the Feature Envy: if the suggestion proposed by the refactoring tool is correct, then an instance of the Feature Envy smell is present in the source code.

```
1  extractMoveMethodRefactoringSuggestions(Method m)
   T = {}
3  S = entity set of m
   for i = 1 to size of S
5    entity = S[i]
     T = T U {entity.ownerClass}
7  sort(T)
   suggestions = {}
9  for i = 1 to size of T
     if(T[i] != m.ownerClass &&
11       modifiesDataStructureInTargetClass(m, T[i]) &&
         preconditionsSatisfied(m, T[i]))
13
           suggestions = suggestions U
15                   {moveMethodSuggestions(m => T[i])}

17   if(suggestions != {})
       return suggestions
19   else
       for i = 1 to size of T
21       if(T[i] = m.ownerClass)
           return {}
23       else if preconditionsSatisfied(m, T[i])
           return moveMethodSuggestions(m => T[i])
25   return {}
```

Listing 2.3: JDeodorant: identification of Move Method Refactoring operations

Tsantalis *et al.* [25] devised a Move Method Refactoring technique integrated in *JDeodorant*. The underlying approach uses a clustering analysis algorithm shown

in Listing 2.3. Given a method M, the approach forms a set T of candidate target classes where M might be moved (set T in Listing 2.3).

This set is obtained by examining the entities (*i.e.*, attributes and methods) that M accesses from the other classes (entity set S in Listing 2.3). In particular, each class in the system containing at least one of the entities accessed by M is added to T. Then, the candidate target classes in T are sorted in descending order according to the number of entities that M accesses from each of them ($sort(T)$ in Listing 2.3). In the subsequent steps each target class T_i is analyzed to verify its suitability to be the recommended class. In particular, T_i must satisfy three conditions to be considered in the set of candidate suggestions: (i) T_i is not the class M currently belongs to, (ii) M modifies at least one data structure in T_i, and (iii) moving M in T_i satisfies a set of behavior preserving preconditions (*e.g.*, the target class does not contain a method with the same signature as M) [25]. The set of classes in T satisfying all the conditions above are put in the suggestions set. If suggestions is not empty, the approach suggests to move M in the first candidate target class following the order of the sorted set T. On the other side, if suggestions is empty, the classes in the sorted set T are again analyzed by applying milder constraints than before. In particular, if a class T_i is the M owner class, then no refactoring suggestion is performed and the algorithm stops. Otherwise, the approach checks if moving the method M into T_i satisfies the behavior preserving preconditions. If so, the approach suggests to move M into T_i. Thus, an instance of the *Feature Envy* smell is identified.

The technique by Tsantalis *et al.* [25] uses structural information to suggest Move Method Refactoring opportunities. However, there are cases where the Feature Envy and the envied class are related by a conceptual linkage rather than a structural one. Here the lexical properties of source code can aid in the identification of the right refactoring to perform. This is the reason why Bavota *et al.* presented MethodBook [30], an approach where methods and classes play the same role of the people and groups, respectively, in FACEBOOK. In particular, methods represent people, and so they have their own information as, for example, method calls or conceptual relationships with the other methods in the same class as well

as the methods in the other classes. To identify the envied class, MethodBook use Relational Topic Model (RTM) [77]. Following the Facebook metaphor, the use of RTM is able to identify "friends" of the method under analysis. If the class having the highest number of "friends" of the considered method is not the current owner class, a refactoring operation is suggested (*i.e.,* a Feature Envy is detected).

The Refused Bequest. In Object Oriented development, one of the key features aiming at reducing the effort and the cost in software maintenance is inheritance [22]. For example, if there is something wrong in the definition of an attribute inherited by some children classes, such attribute needs to be changed only in the parent of such classes. However, it is not uncommon that developers make improper use of the concept of inheritance, especially in the cases where other kind of relationships would be more correct. The *Refused Bequest* smell arises when a subclass does not support the interface of the superclass [22]. On one hand, this happens when a subclass overrides a lot of methods inherited by its parent, on the other hand the relationship of inheritance can be wrong also if the subclass does not override the methods inherited from the parent, but never uses them or such methods are never called by the clients of the subclass. In some cases, this smell simply means that the relationship of inheritance is wrong, namely the subclass is not a specialization of the parent.

A simple naive-method to estimate the presence of Refused Bequest smell in a software system is by looking for classes having the following characteristics:

1. The class is in a hierarchy;

2. The class overrides more than θ% of the methods defined by the parent.

However, the method reported above does not take into account the semantics established by an "is-a" relationship between two classes of the system. For this reason, Ligu *et al.* [78] introduced the identification of Refused Bequest using a combination of static source code analysis and dynamic unit test execution. In particular, the approach identifies classes affected by this smell by looking at the classes that really "wants to support the interface of the superclass" [22]. If a class

does not support the behavior of its parent, a Refused Bequest is detected. In order to understand if a method of a superclass is actually called on subclass instances by other classes of the system, Ligu *et al.* [78] intentionally override these methods introducing an error in the new implementation (*e.g.,* division by zero). If there are classes in the system invoking the method, then a failure will occurs. Otherwise, if the execution of all the test cases does not lead to a failure, the inherited superclass methods are never used by the other classes of the system and, thus, an instance of Refused Bequest is found. The approach proposed by Ligu *et al.* has been implemented as an extension of the *JDeodorant* Eclipse plugin [67].

The Functional Decomposition. The concepts of Object Oriented development are not always clear to developers working for the first time using such a paradigm.

```
RULE_CARD: Functional Decomposition {
  RULE: Functional Decomposition
        {ASSOC: associated FROM: mainClass
          ONE TO: aClass MANY}
  RULE: mainClass
        {UNION NoInheritPoly FunctionClass}
  RULE: NoInheritPoly
        {INTER NoInheritance NoPolymorphism}
  RULE: NoInheritance
        {(METRIC: DIT, SUP_EQ, 1, 0)}
  RULE: NoPolymorphism
      {(STRUCT: DIFFERENT_PARAMETER)}
  RULE: FunctionClass
      {(SEMANTIC: CLASSNAME, {Make, Create, Creator,
         Execute, Exec, Compute, Display, Calculate}}
  RULE: aClass
      {INTER ClassOneMethod FieldPrivate}
  RULE: ClassOneMethod
      {(STRUCT: ONE_METHOD)}
  RULE: FieldPrivate
      {(STRUCT: PRIVATE_FIELD, 100)};
```

Listing 2.4: Rule Card used by DECOR for detecting Functional Decomposition instances.

Indeed, a developer with high experience in functional paradigm tends to apply procedural rules in the development of Object Oriented software, producing errors in the design of the application. The smell coined as *Functional Decomposition* is the most common code smell appearing in these cases. Brown *et al.* [61] define the smell as "a main routine that calls many subroutines". Specifically, it often arises in classes that poorly use inheritance and polymorphism, declaring

many private fields and implementing few methods [61]. In order to detect this smell, some heuristics have been proposed. For example, knowing that usually a routine is called with a name invoking its function, it is not surprising to find an instance of Functional Decomposition in classes called with prefix as *Make* or *Execute*. At the same time, also heuristics based on the structure of a class can support the identification of this smell. For instance, a functional class can have many dependencies with classes composed by a very few number of methods addressing a single function.

The unique approach able to identify this design flaw is DECOR [20]. To infer the presence of the smell, DECOR uses a set of structural properties together to lexical analysis of the name of a class (see the rule card in Listing 2.4). A class is smelly if it is a main class (a class generally characterized by a procedural name, (*e.g., Display*), in which inheritance and polymorphism are poorly used) having many dependencies with small classes (classes with a very few number of methods and many private fields) [20].

2.1.3 Detecting Frequently Changed Classes

Classes that change too frequently threat the stability of the source code and increase the likelihood to introduce bugs [7]. In this category, two code smells from the Fowler's catalogue have been studied, *i.e., Divergent Change* and *Shotgun Surgery*.

The Divergent Change. Fowler describes a *Divergent Change* as a class that is *"commonly changed in different ways for different reasons"* [8]. Classes affected by this smell generally have low cohesion.

The definition of Divergent Change provided by Fowler suggests that structural techniques are not completely suitable to detect instances of such a smell (as demonstrated in Chapter 7 of this thesis). The reason is that in order to identify Divergent Change instances it is necessary to analyze how the system evolves over time. Only one approach provides an attempt to exploit structural information (*i.e.,* coupling) to identify this smell [79]. In particular, using coupling information

it is possible to build a Design Change Propagation Probability (DCPP) matrix. The DCPP is an $n \times n$ matrix where the generic entry Ai, j is the probability that a design change on the artifact i requires a change also to the artifact j. Such probability is given by the cdegree [80], *i.e.*, an indicator of the number of dependencies between two artifacts. Once the matrix is built, a Divergent Change instance is detected if a column in the matrix contains high values for a particular artifact. In other words, high values on the columns of the matrix correspond to have an high number of artifacts related to the one under analysis and so the probability to have a Divergent Change instance.

The Shotgun Surgery. This smell appears when *"every time you make a kind of change, you have to make a lot of little changes to a lot of different classes"* [79]. As for Divergent Change, also in this case finding a purely structural technique able to provide an accurate detection of this smell is rather challenging (see Chapter 7).

Also in this case, the unique approach based on structural analysis is the one presented by Rao and Raddy [79]. By relying on the same DCPP matrix built for the identification of Divergent Change, the approach detects instances of Shotgun Surgery if a row of the matrix contains high values for an artifact. Indeed, in this case there is high probability that a change involving the artifact impact on more than one artifact.

2.1.4 Detecting Code Clones

Classes that show the same code structure in more than one place in the system are affected by the *Duplicate Code* smell. Even if having duplicate components in source code not always implies a higher effort in the maintenance of a system [81], developers may need to modify several times the same feature when a change request involving such duplicate components is received. The identification of code duplication is very challenging simply because, during the evolution, different copies of a feature suffer different changes and this affect the possibility of the identification of the common functionality provided by different copied features. Generally, there are four types of code clones. The type I clones refer to identical

code fragments that differ from each other only for small variations in comments and layout [82]. Type II clones are instead identical fragments except for variations in identifier names and literal values in addition to Type I differences [82]. In addition to this, Type III clones have syntactically similar fragments that differ at the statement level [82]. In particular, such fragments have statements added, modified, or removed with respect to each other. Finally, type IV clones are syntactically dissimilar fragments that implement the same high-level functionality [82].

In the literature several approaches for clone detection have been proposed. It is worth noting that a semantic detection of clones could be very hard to perform and, in general, this is an undecidable problem [83]. This is the reason why most of the proposed techniques focus their attention on the detection of syntactic or structural similarity of source code.

For instance, a common way to detect clones is to analyze the AST of the given program in order to find matches of sub-trees [84, 85, 86]. Alternatively, Kamiya *et al.* [87] introduced the tool CCFINDER, where a program is divided in lexemes and the token sequences are compared in order to find matches between two subsequences. However, such approaches appear to be ineffective in cases where duplicated code suffers several modifications during its evolution. To mitigate such a problem, Jiang *et al.* [88] introduced DECKARD, a technique able to identify clones using a mix of tree-based and syntactic-based approaches. The process they follow can be summarized as follow:

1. Given a program, a parser translates source code into parse tree;
2. Syntactic trees are processed in order to produce a set of vectors capturing the syntactic information of parse tree;
3. The Euclidean distances are performed. Thus, the vectors are clustered;
4. Heuristics are applied to detect clones.

Graph-based techniques [89, 90, 91, 92, 93] rely on the analysis of the program dependence graph (PDG), an intermediate representation of data and control de-

pendencies [94]. In particular, Gabel *et al.* [89] tried to add in DECKARD semantic information derived from the the PDG. They mapped subgraphs to related structured syntax and then detected clones using DECKARD. Chen *et al.* [93] defined a "geometry characteristic" of dependency graphs to measure the similarity between methods before combining method-level similarities to detect application clones in Android markets.

Other approches include the use of textual analysis [95, 96, 97, 98], in which slight transformations to the source code are applied in order to measure the similarity by comparing sequences of text. As a consequence, these techniques are limited in their ability to recognize two fragments as a clone pair even if the difference between them is as inconsequential as a systematic renaming of identifiers.

To overcome the issues of textual-based technique, token-based approaches [99, 100, 101, 102] relaxed the textual-based rule by operating at a higher level of abstraction. In particular, the lexicon of the source code is analyzed in order to produce a stream of tokens and compare subsequences to detect clones. While these techniques generally improve the recognition power, the token abstraction has a tendency to admit more false positives [82].

Recently, White *et al.* [83] proposed a learning-based approach to model code fragments exploiting empirically-based patterns in structures of terms in code. Specifically, the approach relies on deep learning for automatically linking patterns mined at the lexical level with patterns mined at the syntactic level.

2.1.5 Other Detection Approaches

Besides the approaches mentioned above, other researchers proposed complementary heuristics or new methodologies for detecting a variety of code smells.

Munro [26] presented a metric-based detection technique able to identify instances of two smells, *i.e.*, *Lazy Class* and *Temporary Field*, in the source code. A set of thresholds is applied to some structural metrics able to capture those smells. In the case of *Lazy Class*, the metrics used for the identification are Number of Methods (NOM), LOC, Weighted Methods per Class (WMC), and Coupling Between

Objects (CBO).

Van Emden and Moonen [103] presented JCOSMO, a code smell browser that visualizes the detected smells in the source code. In particular, they focus their attention on two Java programming smells, known as *instanceof* and *typecast*. The first occurs when there are too many *instanceof* operators in the same block of code that make the source code difficult to read and understand. The *typecast* smell appears instead when an object is explicitly converted from one class type into another, possibly performing illegal casting which results in a runtime error.

Ratiu *et al.* [12] proposed to use the historical information of the suspected flawed structure to increase the accuracy of the automatic problem detection. However, it is important to note that in this case the change history information is not exploited to detect code smells (as done in Section 7), but for understanding the persistance and the maintenance effort spent on design problems.

Code smell detection can be also formulated as an optimization problem, as pointed out by Kessentini *et al.* [32] as they presented a technique to detect design defects by following the assumption that what significantly diverges from good design practices is likely to represent a design problem. The advantage of their approach is that it does not look for specific code smells (as most approaches) but for design problems in general. Also, in the reported evaluation, the approach was able to achieve a 95% precision in identifying design defects [32]. Kessentini *et al.* [33] also presented a cooperative parallel search-based approach for identifying code smells instances with an accuracy higher than 85%. Boussaa *et al.* [34] proposed the use of competitive coevolutionary search to code-smell detection problem. In their approach two populations evolve simultaneously: the first generates detection rules with the aim of detecting the highest possible proportion of code smells, whereas the second population generates smells that are currently not detected by the rules of the other population. Sahin *et al.* [35] proposed an approach able to generate code smell detection rules using a bi-level optimization problem, in which the first level of optimization task creates a set of detection rules that maximizes the coverage of code smell examples and artificial code smells generated by the second level. The lower level is instead responsible to maximize the

number of code smells artificially generated. The empirical evaluation shows that this approach achieves an average of more than 85% in terms of precision and recall.

Arcelli Fontana *et al.* [36, 37] suggested the use of learning algorithms to discover code smells, pointing out that a training set composed of one hundred instances is sufficient to reach very high values of accuracy. The same authors defined a structural-based approach, named JCODEODOR, for detecting and filtering code smells [104, 105]. Moreover, such approach is able to prioritize code smells through an intensity index [106], which is based on to what extent structural metrics exceed a give threshold. As shown in Chapter 11, we exploited this intensity index in the context of bug prediction.

Finally, a recent trend is the definition of context-based detection approaches. Morales *et al.* [38] proposed the use of developers' context as a way for improving the practical usefulness of code smell detectors, devising an approach for automatic refactoring of code smells. Sae-Lim *et al.* [39], instead, defined a code smell prioritization tool that mines the issue tracker of a system in order to suggest the refactoring of code smells involved in an issue.

2.2 Literature Review on the Empirical Studies on Code Smells

This Section reports an overview of the empirical studies conducted with the aim of assessing the longevity and the impact of code smells on maintainability.

2.2.1 Evolution of Smells

A first study that takes into account the way the code smells evolve during the evolution of a system has been conducted by Chatzigeorgiou and Manakos [13]. The reported results show that (i) the number of instances of code smells increases during time; and (ii) developers are reluctant to perform refactoring operations in order to remove them. On the same line are the results reported by Peters and

Zaidman [107], who show that developers are often aware of the presence of code smells in the source code, but they do not invest time in performing refactoring activities aimed at removing them. A partial reason for this behavior is given by Arcoverde *et al.* [14], who studied the longevity of code smells showing that they often survive for a long time in the source code. The authors point to the will of avoiding changes to API as one of the main reasons behind this result [14].

The evolution of code smells is also studied by Olbrich *et al.* [108], who analyzed the evolution of two types of code smells, namely *God Class* and *Shotgun Surgery*, showing that there are periods in which the number of smells increases and periods in which this number decreases. They also show that the increase/decrease of the number of instances does not depend on the size of the system. Vaucher *et al.* [109] conducted a study on the evolution of the *God Class* smell, aimed at understanding whether they affect software systems for long periods of time or, instead, are refactored while the system evolves. Their goal is to define a method able to discriminate between *God Class* instances that have been introduced by design and *God Class* instances that were introduced unintentionally.

In a closely related field, Bavota *et al.* [110] analyzed the distribution of unit test smells in 18 software systems providing evidence that they are widely spread, but also that most of the them have a strong negative impact on code comprehensibility. Similarly, Palomba *et al.* [111] conducted a study on the distribution of test smells on a dataset of 110 software systems where test code was automatically generated by the EVOSUITE tool [112]. Their findings confirmed that test smells are widely spread.

Göde [113] investigated to what extent code clones are removed through deliberate operations, finding significant divergences between the code clones detected by existing tools and the ones removed by developers. Bazrafshan and Koschke [114] extended the work by Göde, analyzing whether developers remove code clones using deliberate or accidental modifications, finding that the former category is the most frequent. Kim *et al.* [115] studied the lifetime of code clones, finding that many clones are fixed shortly, while long-lived code clones are not easy to refactor because they evolve independently.

Thummalapenta *et al.* [116] introduced the notion of "late propagation" related to changes that have been propagated across cloned code instances at different times. An important difference between research conducted in the area of clone evolution and code smell evolution is that, differently from other code smells, clone evolution can be seen of the co-evolution of multiple, similar (*i.e.*, cloned) code elements, and such evolution can either be consistent or inconsistent (*e.g.*, due to missing change propagation) [116].

2.2.2 Impact of Smells on Maintenance Properties

Several empirical studies have investigated the impact of code smells on maintenance activities. Abbes *et al.* [18] studied the impact of two types of code smells, namely *Blob* and *Spaghetti Code*, on program comprehension. Their results show that the presence of a code smell in a class does not have an important impact on developers' ability to comprehend the code. Instead, a combination of more code smells affecting the same code components strongly decreases developers' ability to deal with comprehension tasks. The interaction between different smell instances affecting the same code components has also been studied by Yamashita *et al.* [9], who confirmed that developers experience more difficulties in working on classes affected by more than one code smell. The same authors also analyzed the impact of code smells on maintainability characteristics [117]. They identified which maintainability factors are reflected by code smells and which ones are not, basing their results on (i) expert-based maintainability assessments, and (ii) observations and interviews with professional developers. Sjoberg *et al.* [118] investigated the impact of twelve code smells on the maintainability of software systems. In particular, the authors conducted a study with six industrial developers involved in three maintenance tasks on four Java systems. The amount of time spent by each developer in performing the required tasks has been measured through an Eclipse plug-in, while a regression analysis has been used to measure the maintenance effort on source code files having specific properties, including the number of smells affecting them. The achieved results show that smells do

not always constitute a problem, and that often class size impacts maintainability more than the presence of smells.

Lozano *et al.* [15] proposed the use of change history information to better understand the relationship between code smells and design principle violations, in order to assess the severity of design flaws. The authors found that the types of maintenance activities performed over the evolution of the system should be taken into account to focus refactoring efforts. In our study, we point out how particular types of maintenance activities (*i.e.,* enhancement of existing features or implementation of new ones) are generally more associated with code smell introduction. Deligiannis *et al.* [119] performed a controlled experiment showing that the presence of *God Class* smell negatively affects the maintainability of source code. Also, the authors highlight an influence played by these smells in the way developers apply the inheritance mechanism.

Khomh *et al.* [16, 17] demonstrated that the presence of code smells increases the code change proneness. Also, they showed that the code components affected by code smells are more fault-prone with respect to components not affected by any smell [16, 17]. Gatrell and Counsell [44] conducted an empirical study aimed at quantifying the effect of refactoring on change- and fault-proneness of classes. In particular, the authors monitored a commercial C# system for twelve months identifying the refactorings applied during the first four months. They examined the same classes for the second four months in order to determine whether the refactoring results in a decrease of change- and fault-proneness. They also compared such classes with the classes of the system that, during the same time period, have not been refactored. The results revealed that classes subject to refactoring have a lower change- and fault-proneness, both considering the time period in which the same classes were not refactored and classes in which no refactoring operations were applied. Li *et al.* [42] empirically evaluated the correlation between the presence of code smells and the probability that the class contains errors. They studied the post-release evolution process showing that many code smells are positively correlated with class errors. Olbrich *et al.* [108] conducted a study on the *God Class* and *Brain Class* code smells, reporting that these code smells

were changed less frequently and had a fewer number of defects with respect to the other classes. D'Ambros *et al.* [41] also studied the correlation between the *Feature Envy* and *Shotgun Surgery* smells and the defects in a system, reporting no consistent correlation between them.

2.2.3 Empirical Studies on Refactoring

Wang *et al.* [120] conducted a survey with ten industrial developers in order to understand which are the major factors that motivate their refactoring activities. The authors report twelve different factors pushing developers to adopt refactoring practices and classified them in *intrinsic motivators* and *external motivators*. In particular, Intrinsic motivators are those for which developers do not obtain external rewards (for example, an intrinsic motivator is the *Responsibility with Code Authorship*, namely developers want to ensure high quality for their code). Regarding the external motivators, an example is the *Recognitions from Others*, i.e., high technical ability can help the software developers gain recognitions.

Murphy-Hill *et al.* [121] analyzed eight different datasets trying to understand how developers perform refactorings. Examples of the exploited datasets are usage data from 41 developers using the Eclipse environment, data from the Eclipse Usage Collector aggregating activities of 13,000 developers for almost one year, and information extracted from versioning systems. Some of the several interesting findings they found were (i) almost 41% of development activities contain at least one refactoring session, (ii) programmers rarely (almost 10% of the time) configure refactoring tools, (iii) commit messages do not help in predicting refactoring, since rarely developers explicitly report their refactoring activities in them, (iv) developers often perform *floss refactoring*, namely they interleave refactoring with other programming activities, and (v) most of the refactoring operations (close to 90%) are manually performed by developers without the help of any tool.

Kim *et al.* [122] presented a survey performed with 328 Microsoft engineers (of which 83% developers) to investigate (i) when and how they refactor code, (ii)

if automated refactoring tools are used by them and (iii) developers' perception towards the benefits, risks, and challenges of refactoring [122]. The main findings of the study reported that:

- While developers recognize refactoring as a way to improve the quality of a software system, in almost 50% of the cases they do not define refactoring as a behavior-preserving operation;
- The most important symptom that pushes developers to perform refactoring is low readability of source code;
- 51% of developers manually perform refactoring;
- The main benefits that the developers observed from the refactoring were improved readability (43%) and improved maintainability (30%);
- The main risk that developers fear when performing refactoring operations is bug introduction (77%).

Kim *et al.* [122] also reported the results of a quantitative analysis performed on the Windows 7 change history showing that code components refactored over time experienced a higher reduction in the number of inter-module dependencies and post-release defects than other modules. Similar results have been obtained by Kataoka *et al.* [123], which analyzed the history of an industrial software system comparing the classes subject to the application of refactorings with the classes never refactored, finding a decreasing of coupling metrics.

Finally, a number of works have studied the relationship between refactoring and software quality. Bavota *et al.* [124] conducted a study aimed at investigating to what extent refactoring activities induce faults. They show that refactorings involving hierarchies (e.g., *pull down method*) induce faults very frequently. Conversely, other kinds of refactorings are likely to be harmless in practice.

Stroggylos and Spinellis [125] studied the impact of refactoring operations on the values of eight object-oriented quality metrics. Their results show the possible negative effects that refactoring can have on some quality metrics (e.g., increased

value of the LCOM metric). On the same line, Stroullia and Kapoor [126], analyzed the evolution of one system observing a decrease of LOC and NOM (Number of Method) metrics on the classes in which a refactoring has been applied. Szoke *et al.* [127] performed a study on five software systems to investigate the relationship between refactoring and code quality. They show that small refactoring operations performed in isolation rarely impact software quality. On the other side, a high number of refactoring operations performed in block helps in substantially improving code quality. Alshayeb [128] investigated the impact of refactoring operations on five quality attributes, namely adaptability, maintainability, understandability, reusability, and testability. Their findings highlight that benefits brought by refactoring operations on some code classes are often counterbalanced by a decrease of quality in some other classes. Moser *et al.* [129] conducted a case study in an industrial environment aimed at investigating the impact of refactoring on the productivity of an agile team and on the quality of the code they produce. The achieved results show that refactoring not only increases software quality but also helps to increase developers' productivity.

PART 2

EMPIRICAL STUDIES ON CODE SMELLS

Chapter 3

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

3.1 Introduction

The technical debt metaphor introduced by Cunningham [4] well explains the trade-offs between delivering the most appropriate but still immature product, in the shortest time possible [4, 130, 131, 132, 133]. Bad code smells (shortly "code smells" or "smells"), *i.e.*, symptoms of poor design and implementation choices [8], represent one important factor contributing to technical debt, and possibly affecting the maintainability of a software system [131]. In the past and, most notably, in recent years, several studies investigated the relevance that code smells have for developers [50, 134], the extent to which code smells tend to remain in a software system for long periods of time [14, 13, 15, 12], as well as the side effects of code smells, such as an increase in change- and fault-proneness [16, 17] or decrease of software understandability [18] and maintainability [19, 10, 9]. While the repercussions of code smells on software quality have been empirically proven, there is still noticeable lack of empirical evidence related to how, when, and why they occur in software projects, as well as whether, after how long, and how they

are removed [130]. This represents an obstacle for an effective and efficient management of technical debt. Also, understanding the typical life-cycle of code smells and the actions undertaken by developers to remove them is of paramount importance in the conception of recommender tools for developers' support. In other words, only a proper undertanding of the phenomenon would allow the creation of recommenders able to highlight the presence of code smells and suggesting refactorings only when appropriate, hence avoiding information overload for developers [135].

Common wisdom suggests that urgent maintenance activities and pressure to deliver features while prioritizing time-to-market over code quality are often the causes of such smells. Generally speaking, software evolution has always been considered as one of the reasons behind "software aging" [5] or "increasing complexity" [3, 136, 137]. Also, one of the common beliefs is that developers remove code smells from the system by performing refactoring operations. However, to the best of our knowledge, there is no comprehensive empirical investigation into *when* and *why* code smells are introduced in software projects, how long they *survive*, and *how they are removed*.

In this chapter we fill the void in terms of our understanding of code smells, reporting the results of a large-scale empirical study conducted on the change history of 200 open source projects belonging to three software ecosystems, namely Android, Apache and Eclipse. The study aims at investigating (i) *when* smells are introduced in software projects, (ii) *why* they are introduced (*i.e.,* under what circumstances smell introductions occur and who are the developers responsible for introducing smells), (iii) *how long they survive* in the system, and (iv) *how they are removed*. To address these research questions, we developed a metric-based methodology for analyzing the evolution of code entities in change histories of software projects to determine when code smells start manifesting themselves and whether this happens suddenly (*i.e.,* because of a pressure to quickly introduce a change), or gradually (*i.e.,* because of medium-to-long range design decisions). We mined over half a million of commits and we manually analyzed over 10K of them to understand how code smells are introduced and removed from software

systems. We are unaware of any published technical debt, in general, and code smells study, in particular, of comparable size. The obtained results allowed us to report quantitative and qualitative evidence on when and why smells are introduced and removed from software projects as well as implications of these results, often contradicting common wisdom. In particular, our main findings show that (i) most of the code smells are introduced when the (smelly) code artifact is created in the first place, and not as the result of maintenance and evolution activities performed on such an artifact, (ii) 80% of code smells, once introduced, are not removed by developers, and (iii) the 20% of removed code smells are very rarely (in 9% of cases) removed as a direct consequence of refactoring activities.

The chapter makes the following notable contributions:

1. *A methodology for identifying smell-introducing changes*, namely a technique able to analyze change history information in order to detect the commit, which introduced a code smell;

2. *A large-scale empirical study involving three popular software ecosystems* aimed at reporting quantitative and qualitative evidence on when and why smells are introduced in software projects, what is their survivability, and how code smells are removed from the source code, as well as implications of these results, often contradicting common wisdom.

3. *A publicly available comprehensive dataset* [138] that enables others to conduct further similar or different empirical studies on code smells (as well as completely reproducing our results).

Implications of the study. From a purely empirical point of view, the study aims at confirming and/or contradicting the common wisdom about software evolution and manifestation of code smells. From a more practical point of view, the results of this study can help distinguish among different situations that can arise in software projects, and in particular in cases where:

- Smells are introduced when a (sub) system has been conceived. Certainly, in such cases smell detectors can help identify potential problems, although this situation can trigger even more serious alarms related to potentially poor design choices made in the system since its inception (*i.e.,* technical debt that smell detectors will not be able to identify from a system's snapshot only), that may require careful re-design in order to avoid worse problems in future.

- Smells occur suddenly in correspondence to a given change, pointing out cases for which recommender systems may warn developers of emergency maintenance activities being performed and the need to consider refactoring activities whenever possible.

- The symptom simply highlights—as also pointed out in a previous study [50, 134]—the intrinsic complexity, size (or any other smell-related characteristics) of a code entity, and there is little or nothing one can do about that. Often some situations that seem to fall in the two cases above should be considered in this category instead.

- Smells manifest themselves gradually. In such cases, smell detectors can identify smells only when they actually manifest themselves (*e.g.,* some metrics go above a given threshold) and suggest refactoring actions. Instead, in such circumstances, tools monitoring system evolution and identifying metric trends, combined with history-based smell detectors [52], should be used.

In addition, our findings, which are related to the very limited refactoring actions undertaken by developers to remove code smells, call for further studies aimed at understanding the reasons behind this result. Indeed, it is crucial for the research community to study and understand whether:

- developers perceive (or don't) the code smells as harmful, and thus they simply do not care about removing them from the system; and/or

Table 3.1: Characteristics of ecosystems under analysis.

Ecosystem	#Proj.	#Classes	KLOC	#Commits	#Issues	Mean Story Length	Min-Max Story Length
Apache	100	4-5,052	1-1,031	207,997	3,486	6	1-15
Android	70	5-4,980	3-1,140	107,555	1,193	3	1-6
Eclipse	30	142-16,700	26-2,610	264,119	124	10	1-13
Overall				**579,671**	**4,803**	**6**	**1-15**

- developers consider the cost of refactoring code smells too high when considering possible side effects (*e.g.,* bug introduction [124]) and expected benefits; and/or
- the available tools for the identification/refactoring of code smells are not sufficient/effective/usable from the developers' perspective.

3.2 Study Design

The *goal* of the study is to analyze the change history of software projects with the *purpose* of investigating when code smells are introduced and fixed by developers and the circumstances and reasons behind smell appearances.

More specifically, the study aims at addressing the following four research questions (RQs):

- **RQ1:** *When are code smells introduced?* This research question aims at investigating to what extent the common wisdom suggesting that "*code smells are introduced as a consequence of continuous maintenance and evolution activities performed on a code artifact*" [8] applies. Specifically, we study "when" code smells are introduced in software systems, to understand whether smells are introduced as soon as a code entity is created, whether smells are suddenly introduced in the context of specific maintenance activities, or whether, instead, smells appear "gradually" during software evolution. To this aim, we investigated the presence of possible trends in the history of code artifacts that characterize the introduction of specific types of smells.

- **RQ2:** *Why are code smells introduced?* The second research question aims at empirically investigating under which circumstances developers are more prone to introduce code smells. We focus on factors that are indicated as possible causes for code smell introduction in the existing literature [8]: the *commit goal* (*e.g.,* is the developer implementing a new feature or fixing a bug?), the *project status* (*e.g.,* is the change performed in proximity to a major release deadline?), and the *developer status* (*e.g.,* a newcomer or a senior project member?).

- **RQ3:** *What is the survivability of code smells?* In this research question we aim to investigate how long a smell remains in the code. In other words, we want to study the *survivability* of code smells, that is the probability that a code smell instance survives over time. To this aim, we employ a statistical method called survival analysis [139]. In this research question, we also investigate differences of survivability among different types of code smells.

- **RQ4:** *How do developers remove code smells?* The fourth and last research question aims at empirically investigating whether and how developers remove code smells. In particular, we want to understand whether code smells are removed using the expected and suggested refactoring operations for each specific type of code smell (as suggested by Fowler [8]), whether they are removed using "unexpected refactorings", or whether such a removal is a side effect of other changes. To achieve this goal, we manually analyzed 979 commits removing code smells by following an open coding process inspired by grounded theory [140].

3.2.1 Context Selection

The *context* of the study consists of the change history of 200 projects belonging to three software ecosystems, namely Android, Apache, and Eclipse. Table 3.1 reports for each of them (i) the number of projects analyzed, (ii) size ranges in terms of the number of classes and KLOC, (iii) the overall number of commits

and issues analyzed, and (iv) the average, minimum, and maximum length of the projects' history (in years) analyzed in each ecosystem. All the analyzed projects are hosted in *Git* repositories and have associated issue trackers.

The Android ecosystem contains a random selection of 70 open source apps mined from the F-Droid[1] forge. The Apache ecosystem consists of 100 Java projects randomly selected among those available[2]. Finally, the Eclipse ecosystem consists of 30 projects randomly mined from the list of GitHub repositories managed by the Eclipse Foundation[3]. The choice of the ecosystems to analyze is not random, but rather driven by the motivation to consider projects having (i) different sizes, *e.g.*, Android apps are by their nature smaller than projects in Apache's and Eclipse's ecosystems, (ii) different architectures, *e.g.*, we have Android mobile apps, Apache libraries, and plug-in based architectures in Eclipse projects, and (iii) different development bases, *e.g.*, Android apps are often developed by small teams whereas several Apache projects are carried out by dozens of developers [141]. Also, we limited our study to 200 projects since, as it will be shown later, the analysis we performed is not only computationally expensive, but also requires the manual analysis of thousands of data points. To sum up, we mined 579,671 commits and 4,803 issues.

We focus our study on the following types of smells:

1. *Blob Class*: a large class with different responsibilities that monopolizes most of the system's processing [61];

2. *Class Data Should be Private*: a class exposing its attributes, violating the information hiding principle [8];

3. *Complex Class*: a class having a high cyclomatic complexity [61];

4. *Functional Decomposition*: a class where inheritance and polymorphism are poorly used, declaring many private fields and implementing few methods [61];

[1] `https://f-droid.org/`
[2] `https://projects.apache.org/indexes/quick.html`
[3] `https://github.com/eclipse`

Table 3.2: Quality metrics measured in the context of **RQ1.**

Metric	Description
Lines of Code (LOC)	The number of lines of code excluding white spaces and comments
Weighted Methods per Class (WMC) [64]	The complexity of a class as the sum of the McCabe's cyclomatic complexity of its methods
Response for a Class (RFC) [64]	The number of distinct methods and constructors invoked by a class
Coupling Between Object (CBO) [64]	The number of classes to which a class is coupled
Lack of COhesion of Methods (LCOM) [64]	The higher the pairs of methods in a class sharing at least a field, the higher its cohesion
Number of Attributes (NOA)	The number of attributes in a class
Number of Methods (NOM)	The number of methods in a class

5. *Spaghetti Code*: a class without structure that declares long methods without parameters [61].

While several other smells exist in the literature [8, 61], we need to limit our analysis to a subset due to computational constraints. However, we carefully keep a mix of smells related to complex/large code components (*e.g.,* Blob Class, Complex Class) as well as smells related to the lack of adoption of good Object-Oriented coding practices (*e.g.,* Class Data Should be Private, Functional Decomposition). Thus, the considered smells are representative of the categories of smells investigated in previous studies (see Section 2).

3.2.2 Data Extraction and Analysis

This subsection describes the data extraction and analysis process that we followed to answer our research questions.

When are code smells introduced?

To answer **RQ1** we firstly clone the 200 GIT repositories. Then, we analyze each repository r_i using a tool that we developed (named as HISTORYMINER), with the purpose of identifying smell-introducing commits. Our tool mines the entire change history of r_i, checks out each commit in chronological order, and runs an implementation of the *DECOR* smell detector based on the original rules defined by Moha *et al.* [20]. *DECOR* identifies smells using detection rules based on the

values of internal quality metrics[4]. The choice of using *DECOR* is driven by the fact that (i) it is a state-of-the-art smell detector having a high accuracy in detecting smells [20]; and (ii) it applies simple detection rules that allow it to be very efficient. Note that we ran *DECOR* on all source code files contained in r_i only for the first commit of r_i. In the subsequent commits *DECOR* has been executed only on code files added or modified in each specific commit to save computational time. As an output, our tool produces, for each source code file $f_j \in r_i$ the list of commits in which f_j has been involved, specifying if f_j has been added, deleted, or modified and if f_j was affected in that specific commit, by one of the five considered smells.

Starting from the data generated by the HISTORYMINER, we compute, for each type of smell ($smell_k$) and for each source code file (f_j), the number of commits performed on f_j since the first commit involving f_j and adding the file to the repository, up to the commit in which *DECOR* detects that f_j is affected by $smell_k$. Clearly, such numbers are only computed for files identified as affected by the specific $smell_k$.

When analyzing the number of commits needed for a smell to affect a code component, we can have two possible scenarios. In the first scenario, smell instances are introduced during the creation of source code artifacts, *i.e.*, in the first commit involving a source code file. In the second scenario, smell instances are introduced after several commits and, thus, as a result of multiple maintenance activities. For the latter scenario, besides running the *DECOR* smell detector for the project snapshot related to each commit, the HISTORYMINER also computes, for each snapshot and for each source code artifact, a set of quality metrics (see Table 3.2). As done for *DECOR*, quality metrics are computed for all code artifacts only during the first commit, and updated at each subsequent commit for added and modified files. The purpose of this analysis is to understand whether the trend followed by such metrics differ between files affected by a specific type of smell and files not affected by such a smell. For example, we expect that classes

[4]An example of detection rule exploited to identify Blob classes can be found at `http://tinyurl.com/paf9gp6`.

becoming Blobs will exhibit a higher growth rate than classes that are not going to become Blobs.

In order to analyze the evolution of the quality metrics, we need to identify the function that best approximates the data distribution, *i.e.,* the values of the considered metrics computed in a sequence of commits. We found that the best model is the linear function. Note that we only consider linear regression models using a single metric at a time (*i.e.,* we did not consider more than one metric in the same regression model) since our interest is to observe how a single metric in isolation describes the smell-introducing process. We consider the building of more complex regression models based on more than one metric as part of our future work.

Having identified the model to be used, we compute, for each file $f_j \in r_i$, the regression line of its quality metric values. If file f_j is affected by a specific $smell_k$, we compute the regression line considering the quality metric values computed for each commit involving f_j from the first commit (*i.e.,* where the file was added to the versioning system) to the commit where the instance of $smell_k$ was detected in f_j. Instead, if f_j is not affected by any smell, we consider only the first n^{th} commits involving the file f_j, where n is the average number of commits required by $smell_k$ to affect code instances. Then, for each metric reported in Table 3.2, we compare the distributions of regression line slopes for smell-free and smelly files. The comparison is performed using a two-tailed Mann-Whitney U test [142]. The results are intended as statistically significant at $\alpha = 0.05$. We also estimate the magnitude of the observed differences using the Cliff's Delta (or d), a non-parametric effect size measure [143] for ordinal data. We follow the guidelines in [143] to interpret the effect size values: small for $d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$.

Overall, the data extraction for **RQ1** (*i.e.,* the smells detection and metrics computation at each commit for the 200 systems) took eight weeks on a Linux server having 7 quad-core 2.67 GHz CPU (28 cores) and 24 Gb of RAM.

Why are code smells introduced?

One challenge arising when answering **RQ2** is represented by the identification of the specific commit (or also possibly a set of commits) where the smell has been introduced (from now on referred to as a *smell-introducing commit*). Such information is crucial to explain under which circumstances these commits were performed. A trivial solution would have been to use the results of our **RQ1** and consider the commit c_s in which *DECOR* detects for the first time a smell instance $smell_k$ in a source code file f_j as a commit-introducing smell in f_j. However, while this solution would work for smell instances that are introduced in the first commit involving f_j (there is no doubt on the commit that introduced the smell), it would not work for smell instances that are the consequence of several changes, performed in n different commits involving f_j. In such a situation, on one hand, we cannot simply assume that the first commit in which *DECOR* identifies the smell is the one introducing that smell, because the smell appearance might be the result of several small changes performed across the n commits. On the other hand, we cannot assume that all n commits performed on f_j are those (gradually) introducing the smell, since just some of them might have pushed f_j toward a smelly direction. Thus, to identify the smell-introducing commits for a file f_j affected by an instance of a smell ($smell_k$), we use the following heuristic:

- **if** $smell_k$ has been introduced in the commit c_1 where f_j has been added to the repository, **then** c_1 is the smell-introducing commit;
- **else** given $C = \{c_1, c_2, \ldots, c_n\}$ the set of commits involving f_j and leading to the detection of $smell_k$ in c_n we use the results of **RQ1** to select the set of quality metrics M allowing to discriminate between the groups of files that are affected and not affected in their history by $smell_k$. These metrics are those for which we found statistically significant difference between the slope of the regression lines for the two groups of files accompanied by at least a medium effect size. Let s be the slope of the regression line for the metric $m \in M$ built when considering all commits leading f_j to become affected by a smell and s_i the slope of the regression line for the metric m built when con-

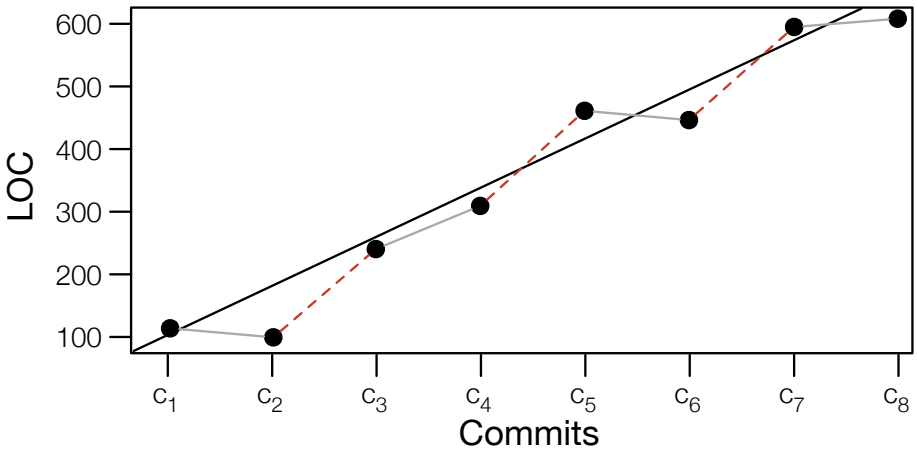


Figure 3.1: Example of identifying smell-introducing commits.

sidering just two subsequent commits, *i.e.*, c_{i-1} and c_i for each $i \in [2, ..., n]$. A commit $c_i \in C$ is considered as a smell-introducing commit if $|s_i| > |s|$, *i.e.*, the commit c_i significantly contributes to the increment (or decrement) of the metric m.

Figure 3.1 reports an example aimed at illustrating the smell-introducing commits identification for a file f_j. Suppose that f_j has been involved in eight commits (from c_1 to c_8), and that in c_8 a Blob instance has been identified by *DECOR* in f_j. Also, suppose that the results of our **RQ1** showed that the LOC metric is the only one "characterizing" the Blob introduction, *i.e.*, the slope of the LOC regression line for Blobs is significantly different than the one of the regression line built for classes which are not affected by the Blob smell. The black line in Figure 3.1 represents the LOC regression line computed among all the involved commits, having a slope of 1.3. The gray lines represent the regression lines between pairs of commits (c_{i-1}, c_i), where c_i is not classified as a smell-introducing commit (their slope is lower than 1.3). Finally, the red-dashed lines represent the regression lines between pairs of commits (c_{i-1}, c_i), where c_i is classified as a smell-introducing commit (their slope is higher than 1.3). Thus, the smell-introducing commits in the example depicted in Figure 3.1 are: c_3, c_5, and c_7. While other commits may possibly contribute to the smell introduction, our methodology identifies as smell-

Table 3.3: Tags assigned to the smell-introducing commits.

Tag	Description	Values
	COMMIT GOAL TAGS	
Bug fixing	The commit aimed at fixing a bug	[true,false]
Enhancement	The commit aimed at implementing an enhancement in the system	[true,false]
New feature	The commit aimed at implementing a new feature in the system	[true,false]
Refactoring	The commit aimed at performing refactoring operations	[true,false]
	PROJECT STATUS TAGS	
Working on release	The commit was performed [value] before the issuing of a major release	[one day, one week, one month, more than one month]
Project startup	The commit was performed [value] after the starting of the project	[one week, one month, one year, more than one year]
	DEVELOPER STATUS TAGS	
Workload	The developer had a [value] workload when the commit has been performed	[low,medium,high]
Ownership	The developer was the owner of the file in which the commit introduced the smell	[true,false]
Newcomer	The developer was a newcomer when the commit was performed	[true,false]

introducing commits the ones that are more likely to contain modifications having a strong negative impact on the overall quality of a class (*e.g.*, the commits in which the size of a class increases a lot). Overall, we obtained 9,164 smell-introducing commits in 200 systems, that we used to answer **RQ2**.

After having identified smell-introducing commits, with the purpose of understanding *why* a smell was introduced in a project, we classify them by assigning to each commit one or more tags among those reported in Table 3.3. The first set of tags (*i.e.*, commit goal tags) aims at explaining *what the developer was doing when introducing the smell*. To assign such tags we firstly download the issues for all 200 projects from their JIRA or BUGZILLA issue trackers. Then, we check whether any of the 9,164 smell-introducing commits were related to any of the collected issues. To link issues to commits we used (and complemented) two ex-

isting approaches. The first one is the regular expression-based approach by Fischer *et al.* [144] matching the issue ID in the commit note. The second one is a re-implementation of the *ReLink* approach proposed by Wu *et al.* [145], which considers the following constraints: (i) matching the committer/authors with issue tracking contributor name/email; (ii) the time interval between the commit and the last comment posted by the same author/contributor on the issue tracker must be less than seven days; and (iii) the Vector Space Model (VSM) [55] cosine similarity between the commit note and the last comment referred above must be greater than 0.7. RELINK has been shown to accurately link issues and commits (89% for precision and 78% for recall) [145]. When it was possible to identify a link between one of the smell-introducing commits and an issue, and the issue type was one of the goal-tags in our design (*i.e.,* bug, enhancement, or new feature), such tag was automatically assigned to the commit and its correctness was manually double-checked[5], which verified the correctness of the issue category (*e.g.,* that an issue classified as a bug was actually a bug). We were able to automatically assign a tag with this process in 471 cases, *i.e.,* for a small percentage (5%) of the commits, which is not surprising and in agreement with previous findings [146]. Of these 471 were automatically assigned tags, 126 were corrected during the manual double-check, most of them (96) due to a misclassification between enhancement and new feature. In the remaining 8,693 cases, two of the authors[6] manually analyzed the commits, assigning one or more of the goal-tags by relying on the analysis of the commit messages and of the unix diffs between the commit under analysis and its predecessor.

Concerning the project-status tags (see Table 3.3), the *Working on release* tag can assume as possible values *one day*, *one week*, *one month*, or *more than one month* before the issuing of a major release. The aim of such a tag is to indicate whether, *when introducing the smell, the developer was close to a project's deadline*. We just consider major releases since those are the ones generally representing a real deadline for developers, while minor releases are sometimes issued just due to a single bug

[5]The thesis' author was responsible of this task.

[6]The thesis' author was equally involved in this task.

fix. To assign such tags, one of the authors[7] identified the dates in which the major releases were issued by exploiting the `Git` tags (often used to tag releases), and the commit messages left by developers. Concerning the *Project startup* tag, it can assume as values *one week*, *one month*, *one year*, or *more than one year* after the project's start date. This tag can be easily assigned by comparing the commit date with the date in which the project started (*i.e.,* the date of the first commit). This tag can be useful to verify whether *during the project's startup, when the project design might not be fully clear, developers are more prone to introduce smells*. Clearly, considering the date of the first commit in the repository as the project's startup date can introduce imprecisions in our data in case of projects migrated to `Git` in a later stage of their history. For this reason, we verify whether the first release of each project in our dataset was tagged with 0.1 or 1.0 (*i.e.,* a version number likely indicating the first release of a project). As a result, we exclude from the *Project startup* analysis 31 projects having a partial change history in the mined `Git` repository, for a total of 552 smell-introducing commits excluded. While we acknowledge that also this heuristic might introduce imprecisions (*e.g.,* a project starting from release 1.0 could still have a previous $0.x$ release), we are confident that it helps in eliminating most of the problematic projects from our dataset.

Finally, we assign developer-status tags to smell-introducing commits. The *Workload* tag measures how busy a developer was when introducing the bad smell. In particular, we measure the *Workload* of each developer involved in a project using time windows of one month, starting from the date in which the developer joined the project (*i.e.,* performed the first commit). The *Workload* of a developer during one month is measured in terms of the number of commits she performed in that month. We are aware that such a measure (i) is an approximation because different commits can require different amount of work; and (ii) a developer could also work on other projects. When analyzing a smell-introducing commit performed by a developer d during a month m, we compute the workload distribution for all developers of the project at m. Then, given Q_1 and Q_3, the first and the third quartile of such distribution, respectively, we assign: *low* as *Workload* tag if

[7]The thesis' author was responsible of this task.

the developer performing the commit had a workload lower than Q_1, *medium* if $Q_1 \leq workload < Q_3$, *high* if the workload was higher than Q_3.

The *Ownership* tag is assigned if the developer performing the smell-introducing commit is the owner of the file on which the smell has been detected. As defined by Bird *et al.* [147], a file owner is a developer responsible for more than 75% of the commits performed on the file. Lastly, the *Newcomer* tag is assigned if the smell-introducing commit falls among the first three commits in the project for the developer responsible for it.

After assigning all the described tags to each of the 9,164 smell-introducing commits, we analyzed the results by reporting descriptive statistics of the number of commits to which each tag type has been assigned. Also, we discuss several qualitative examples helping to explain our findings.

What is the survivability of code smells?

To address **RQ3**, we need to determine when a smell has been introduced and when a smell disappears from the system. To this aim, given a file f, we formally define two types of commits:

1. *last-smell-introducing commit*: A commit c_i modifying a file f such that, f is affected by a code smell $smell_k$ after commit c_i while it was not affected by $smell_k$ before c_i. Even if an artifact can become smelly as consequence of several modifications (see **RQ2**), in this analysis we are interested in finding a specific date in which an artifact can actually be considered smelly. To this aim we consider the latest possible commit before f actually becomes smelly. Clearly, when a smell is introduced gradually, this commit is not the only responsible for the smell introduction, but, rather, it represents the "turning point" of the smell introduction process.

2. *smell-removing commit*: A commit c_i modifying a file f such that f is not affected by a code smell $smell_k$ after c_i while it was affected by $smell_k$ before c_i. Also, in this case, it may happen that the smell can be gradually removed,

though we take the first commit in which the code smell detector does not spot the smell anymore.

Based on what has been discussed above, given a code smell $smell_k$, the time interval between the *last-smell-introducing commit* and the *smell-removing commit* is defined as *smelly interval*, and determines the longevity of $smell_k$. Given a smelly interval for a code smell affecting the file f and bounded by the *last-smell-introducing commit* c_i and the *smell-removing commit* c_j, we compute as proxies for the smell longevity:

- #*days*: the number of days between the introduction of the smell ($c_i.time$) and its fix ($c_j.time$);

- #*commits*: the number of commits between c_i and c_j that modified the artifact f.

These two proxies provide different and complementary views about the survivability of code smells. Indeed, considering only the #*days* (or any other time-based proxy) could lead to misleading interpretations in cases in which a project is mostly inactive (*i.e.*, no commits are performed) in a given time period. For example, suppose that a smell instance $smell_k$ is refactored 10 months after its introduction in the system. The #*days* proxy will indicate a very high survivability ($\sim$300 days) for $smell_k$. However, we do not know whether the project was active in such a time period (and thus, if developers actually had the chance to fix $smell_k$). The #*commits* will provide us with such information: If the project was active, it will concur with the #*days* proxy in indicating a high survivability for $smell_k$, otherwise it will "contradict", showing a low survivability in terms of #*commits*.

Since we are analyzing a finite change history for a given repository, it could happen that for a specific file and a smell affecting it we are able to detect the *last-smell-introducing commit* but not the *smell-removing commit*, due to the fact that the file is still affected by the code smell in the last commit we analyzed. In other words, we can discriminate two different types of smelly intervals in our dataset:

1. *Closed Smelly Intervals*: intervals delimited by a *last-smell-introducing commit* as well as by a *smell-removing commit*;

2. *Censored Smelly Intervals*: intervals delimited by a *last-smell-introducing commit* and by the end of the change history (*i.e.,* the date of the last commit we analyzed).

In total, we identified 1,426 closed smelly intervals and 9,197 censored smelly intervals. After having collected this data, we answer **RQ3** by relying on *survival analysis* [139], a statistical method that aims at analyzing and modeling the time duration until one or more events happen. Such time duration is modeled as a random variable and typically it has been used to represent the time to the failure of a physical component (mechanical or electrical) or the time to the death of a biological unit (patient, animal, cell, *etc.*) [139]. The survival function $S(t) = Pr(T > t)$ indicates the probability that a subject (in our case the code smell) survives longer than some specified time t. The survival function never increases as t increases; also, it is assumed $S(0) = 1$ at the beginning of the observation period, and, for time $t \to \infty$, $S(\infty) \to 0$. The goal of the survival analysis is to estimate such a survival function from data and assess the relationship of explanatory variables (covariates) to survival time. Time duration data can be of two types:

1. *Complete data*: the value of each sample unit is observed or known. For example, the time to the failure of a mechanical component has been observed and reported. In our case, the code smell disappearance has been observed.

2. *Censored Data*: The event of interest in the analysis has not been observed yet (so it is considered as unknown). For example, a patient cured with a particular treatment has been alive till the end of the observation window. In our case, the smell remains in the system until the end of the observed project history. For this sample, the time-to-death observed is a censored value, because the event (death) has not occurred during the observation.

Both complete and censored data can be used, if properly marked, to generate a survival model. The model can be visualized as a survival curve that shows the

survival probability as a function of the time. In the context of our analysis, the population is represented by the code smell instances while the event of interest is its fix. Therefore, the "time-to-death" is represented by the observed time from the introduction of the code smell instance, till its fix (if observed in the available change history). We refer to such a time period as "the lifetime" of a code smell instance. Complete data is represented by those instances for which the event (fix) has been observed, while censored data refers to those instances which have not been fixed in the observable window. We generate survival models using both the #days and #commits in the smelly intervals as time variables. We analyzed the survivability of code smells by ecosystem. That is, for each ecosystem, we generated a survival model for each type of code smell by using R and the `survival` package[8]. In particular, we used the `Surv` type to generate a survival object and the `survfit` function to compute an estimate of a survival curve, which uses Kaplan-Meier estimator [148] for censored data. In the latter, we use the `conf.type=''none''` argument to specify that we do not want to include any confidence interval for the survival function. Also, we decided to use the Kaplan-Meier estimator, a non-parametric survival analysis method, since we cannot assume a particular distribution of survival times. Such an estimator has been widely used in the literature, for example to study the longevity of Debian packages [149] or to analyze when source code becomes dead code (unused code) [150].

We report the survival function for each type of code smell grouped by ecosystem. In addition, we compare the survival curve of artifacts that are born smelly (*i.e.*, those in which the code smell appears in the commit creating the artifact) with the survival curve of artifacts that became smelly during maintenance and evolution activities.

It is important to highlight that, while the survival analysis is designed to deal with censored data, we perform a cleaning of our dataset aimed at reducing possible biases caused by censored intervals before running the analysis. In particular, code smell instances introduced too close to the end of the observed change history can potentially influence our results, since in these cases the period of time

[8] `https://cran.r-project.org/package=survival`

needed for their removal is too small for being analyzed. Thus, we excluded from our survival analysis all censored intervals for which the *last-smell-introducing commit* was "too close" to the last commit we analyzed in the system's change history (*i.e.*, for which the developers did not have "enough time" to fix them). To determine a threshold suitable to remove only the subset of smell instances actually too close to the end of the analyzed change history, we study the distribution of the number of days needed to fix the code smell instance (*i.e.*, the length of the closed smelly interval) in our dataset and, then, we choose an appropriate threshold (see Section 3.3.3).

How do developers remove code smells?

In order to understand how code smells disappear from the system, we manually analyzed a randomly selected set of 979 *smell-removing commits*. Such a set represents a 95% statistically significant stratified sample with a 5% confidence interval of the 1,426 smell-removing commits in our dataset. The *strata* of such a sample are represented by (i) the three ecosystems analyzed (*i.e.*, we make sure to consider a statistically significant sample for each of the three subject ecosystems), and (ii) the five different code smells considered in our study, *i.e.*, the higher the number of fixing commits involving a smell type (*e.g.*, Blob), the higher the number of *smell-removing commits* involving such a smell type in our manually evaluated sample. In other words, we determined a sample size (for the desired confidence interval and significance level) for each combination of smell type and ecosystem, sampled and manually analyzed accordingly.

To analyze and categorize the type of action performed by the developers that caused the smell to disappear (*e.g.*, refactoring, code deletion, *etc.*), we followed an open coding process. In particular, we randomly distributed the 979 commits among three of the authors of the corresponding paper[9] ($\sim$326 commits each). Each of the involved authors independently analyzed the commits assigned to him by relying on the commit note and the unix diff as shown by GitHub (all sub-

[9]The thesis' author was equally involved in this task.

ject systems are hosted on GitHub). The output of this phase was the assignment of each *smell-removing commit* to a given category explaining why the smell disappeared from the system (*e.g.,* the smell has been refactored, the code affected by the smell has been deleted, *etc.*). Then, the three authors involved in the classification discussed their codings in order to (i) double-check the consistency of their individual categorization, and (ii) refine the identified categories by merging similar categories they identified or splitting when it was the case.

The output of our open coding procedure is the assignment of the 979 commits to a category explaining the reason *why* a specific smell disappeared in a given commit. We quantitatively and qualitatively discuss such data in our results section.

3.3 Analysis of the Results

This section reports the analysis of the results achieved in our study and aims at answering the four research questions formulated in Section 3.2.

3.3.1 When are code smells introduced?

Figure 3.2 shows the distribution of the number of commits required by each type of smell to manifest itself. The results are grouped by ecosystems; also, we report the *Overall* results (all ecosystems together).

As we can observe in Figure 3.2, in almost all the cases the median number of commits needed by a smell to affect code components is zero, except for Blob on Android (median=3) and Complex Class on Eclipse (median=1). In other words, most of the smell instances (at least half of them) are introduced when a code entity is added to the versioning system. This is quite surprising finding, considering the common wisdom that *smells are generally the result of continuous maintenance activities performed on a code component* [8].

However, the box plots also indicate (i) the presence of several outliers; and that (ii) for some smells, in particular Blob and Complex Class, the distribution is

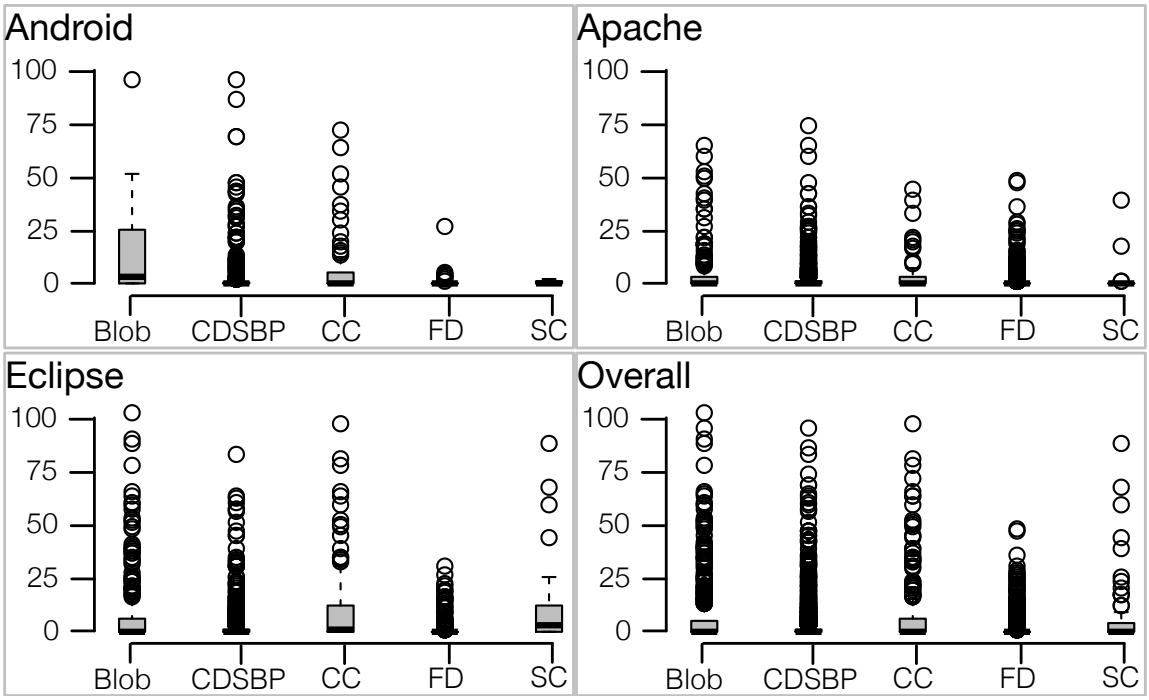


Figure 3.2: The number of commits required by a smell to manifest itself.

quite skewed. This means that besides smell instances introduced in the first commit, there are also several smell instances that are introduced as a result of several changes performed on the file during its evolution. In order to better understand such phenomenon, we analyzed how the values of some quality metrics change during the evolution of such files.

Table 3.4 presents the descriptive statistics (mean and median) of the slope of the regression line computed, for each metric, for both smelly and clean files. Also, Table 3.4 reports the results of the Mann-Whitney test and Cliff's d effect size (**L**arge, **M**edium, or **S**mall) obtained when analyzing the difference between the slope of regression lines for clean and smelly files. Column *cmp* of Table 3.4 shows a $\uparrow$ ($\downarrow$) if for the metric m there is a statistically significant difference in the m's slope between the two groups of files (*i.e., clean* and *smelly*), with the smelly ones exhibiting a higher (lower) slope; a "$-$" is shown when the difference is not statistically significant.

The analysis of the results reveals that for all the smells, but Functional De-

Table 3.4: **RQ1**: slope affected *vs* slope not affected - Mann-Whitney test (adj. p-value) and Cliff's Delta (d).

Ecosys.	Smell	Affected	LOC			LCOM			WMC			RFC			CBO			NOM			NOA		
			mean	med	cmp	mean	med	cmp	mean	med	cmp	mean	med	cmp	mean	med	cmp	mean	med	cmp	mean	med	cmp
Android	Blob	NO	0.68	0		0.55	0		0.17	0		0.13	0		0.15	0		0.07	0		0.09	0	
		YES	32.90	12.51	↑	13.80	2.61	↑	3.78	1.81	↑	5.39	3.47	↑	1.34	0.69	↑	1.15	0.57	↑	0.49	0.13	↑
		p-value	<0.01			<0.01			<0.01			<0.01			<0.01			<0.01			<0.01		
		Cliff's d	0.65 (L)			0.38 (M)			0.53 (L)			0.64 (L)			0.66 (L)			0.51 (L)			0.56 (L)		
	CDSP	NO	0.42	0		0.12	0		0.12	0		0.05	0		0.09	0		0.05	0		0.06	0	
		YES	4.43	1.68	↑	0.83	0	—	0.33	0	—	0.27	0	—	0.36	0.18	↑	0.17	0	—	2.60	0.69	↑
		p-value	<0.01			0.26			0.88			0.86			<0.01			0.71			<0.01		
		Cliff's d	0.45 (M)			0.06 (N)			-0.01 (N)			-0.01 (N)			0.26 (S)			0.02 (N)			0.78 (L)		
	CC	NO	0.67	0		0.48	0		0.19	0		0.14	0		0.15	0		0.08	0		0.09	0	
		YES	7.71	6.81	↑	11.16	4.12	↑	2.61	2.20	↑	2.42	1.01	↑	0.33	0.28	↑	0.67	0.50	↑	0.18	0.10	↑
		p-value	<0.01			<0.01			<0.01			<0.01			<0.01			<0.01			<0.01		
		Cliff's d	0.63 (L)			0.45 (M)			0.76 (L)			0.64 (L)			0.32 (S)			0.67 (L)			0.39 (M)		
	FD	NO	0.99	0		0.62	0		0.29	0		0.31	0		0.40	0		0.11	0		0.11	0	
		YES	-10.56	-1.00	↓	-2.65	0	↓	-2.74	-0.60	↓	-3.49	0	↓	0.78	0.49	—	-1.13	-0.30	↓	-0.91	0	↓
		p-value	<0.01			<0.01			<0.01			0.02			0.09			<0.01			0.01		
		Cliff's d	-0.55 (L)			-0.49 (L)			-0.59 (L)			-0.42 (M)			0.32 (S)			-0.76 (L)			-0.45 (M)		
	SC	NO	1.42	0		0.96	0		0.31	0		0.42	0		0.29	0		0.11	0		0.13	0	
		YES	144.2	31.0	↑	69.17	100.00	↑	10.17	10.00	↑	6.33	5.00	↑	0.67	1.00	—	3	3	↑	0.16	0	↑
		p-value	<0.01			<0.01			<0.01			<0.01			0.50			<0.01			0.04		
		Cliff's d	0.99 (L)			0.98 (L)			0.99 (L)			0.95 (L)			0.22 (S)			0.99 (L)			0.68 (L)		
Apache	Blob	NO	0.40	0		0.42	0		0.13	0		0.13	0		0.05	0		0.05	0		0.03	0	
		YES	91.82	33.58	↑	384.70	12.40	↑	17.79	4.92	↑	27.61	7.09	↑	2.17	0.50	↑	7.64	1.72	↑	0.77	0.05	↑
		p-value	<0.01			<0.01			<0.01			<0.01			<0.01			<0.01			<0.01		
		Cliff's d	0.92 (L)			0.52 (L)			0.77 (L)			0.74 (L)			0.44 (M)			0.82 (L)			0.22 (S)		
	CDSP	NO	0.43	0		0.54	0		0.12	0		0.12	0		0.10	0		0.05	0		0.03	0	
		YES	8.69	2.03	↑	2.44	0	—	0.61	0	—	0.59	0	—	0.55	0.06	↑	0.23	0	—	3.28	1.07	↑
		p-value	<0.01			0.28			0.46			0.45			<0.01			0.37			<0.01		
		Cliff's d	0.63 (L)			-0.04 (N)			-0.03 (N)			0.03 (N)			0.25 (S)			-0.03 (N)			0.86 (L)		
	CC	NO	0.36	0		0.47	0		0.12	0		0.13	0		0.09	0		0.05	0		0.04	0	
		YES	121.80	25.86	↑	886.50	152.40	↑	31.87	10.36	↑	39.81	7.21	↑	3.45	0.53	↑	13.99	3.56	↑	0.17	0	↑
		p-value	<0.01			<0.01			<0.01			<0.01			<0.01			<0.01			0.02		
		Cliff's d	0.81 (L)			0.70 (L)			0.83 (L)			0.74 (L)			0.53 (L)			0.82 (L)			0.23 (S)		
	FD	NO	0.52	0		0.812	0		0.16	0		0.14	0		0.10	0		0.07	0		0.030	0	
		YES	-13.78	-3.32	↓	-5.98	-0.30	↓	-6.16	-1.00	↓	-4.81	-0.52	↓	-0.28	0	↓	-2.82	-0.53	↓	-0.40	0	↓
		p-value	<0.01			<0.01			<0.01			<0.01			<0.01			<0.01			<0.01		
		Cliff's d	-0.72 (L)			-0.46 (M)			-0.66 (L)			-0.49 (L)			-0.14 (N)			-0.67 (L)			-0.35 (M)		
	SC	NO	0.54	0		0.11	0		0.11	0		0.12	0		0.14	0		0.04	0		0.04	0	
		YES	273.00	129.90	↑	232.30	4.50	—	7.09	6.50	↑	10.81	10.15	↑	0.96	0.92	—	3.41	3.00	↑	2.29	2.08	↑
		p-value	<0.01			0.52			<0.01			<0.01			0.12			<0.01			0.02		
		Cliff's d	0.94 (L)			0.17 (S)			0.91 (L)			0.95 (L)			0.44 (M)			0.94 (L)			0.63 (L)		
Eclipse	Blob	NO	0.02	0		0.02	0		-0.01	0		-0.03	0		0.13	0		-0.01	0		0.01	0	
		YES	69.51	28.15	↑	1208.00	14.71	↑	17.10	2.92	↑	18.15	2.44	↑	0.58	0.01	↑	7.11	1.09	↑	3.11	0.09	↑
		p-value	<0.01			<0.01			<0.01			<0.01			<0.01			<0.01			<0.01		
		Cliff's d	0.86 (L)			0.54 (L)			0.76 (L)			0.65 (L)			0.20 (S)			0.75 (L)			0.50 (L)		
	CDSP	NO	0.01	0		0.34	0		<-0.01	0		-0.02	0		0.13	0		<-0.01	0		0.01	0	
		YES	12.58	2.50	↑	749.1	0	↑	2.77	0	↑	0.70	0	↑	0.37	0	—	2.10	0	↑	4.01	1	↑
		p-value	<0.01			<0.01			<0.01			<0.01			0.53			<0.01			<0.01		
		Cliff's d	0.65 (L)			0.13 (N)			0.16 (S)			0.12 (N)			0.03 (N)			0.18 (S)			0.90 (L)		
	CC	NO	0.02	0		0.21	0		-0.01	0		-0.05	0		0.11	0		-0.01	0		0.02	0	
		YES	57.72	18.00	↑	2349.00	141.70	↑	19.86	4.86	↑	10.46	0.82	↑	0.68	0.01	↑	10.23	1.94	↑	3.10	<0.01	↑
		p-value	<0.01			<0.01			<0.01			<0.01			0.02			<0.01			<0.01		
		Cliff's d	0.82 (L)			0.75 (L)			0.84 (L)			0.54 (L)			0.15 (S)			0.83 (L)			0.42 (M)		
	FD	NO	-0.02	0		0.67	0		-0.02	0		-0.02	0		0.13	0		-0.01	0		0.02	0	
		YES	-15.09	-5.40	↓	-5.23	-0.95	↓	-5.15	-1.71	↓	-4.06	-0.60	↓	-0.16	0.16	↑	-2.39	-0.60	↓	-0.35	0	—
		p-value	<0.01			<0.01			<0.01			<0.01			<0.01			<0.01			0.88		
		Cliff's d	-0.72 (L)			-0.61 (L)			-0.71 (L)			-0.51 (L)			0.23 (S)			-0.74 (L)			-0.01 (N)		
	SC	NO	0.07	0		1.19	0		0.02	0		-0.06	0		0.15	0		-0.01	0		0.02	0	
		YES	114.40	42.74	↑	698.4	137.3	↑	16.65	4.03	↑	9.47	0.03	↑	1.37	0	—	6.44	2.39	↑	9.30	1.17	↑
		p-value	<0.01			<0.01			<0.01			<0.01			0.97			<0.01			<0.01		
		Cliff's d	0.92 (L)			0.52 (L)			0.61 (L)			0.44 (M)			0.01 (N)			0.51 (L)			0.65 (L)		
Overall	Blob	NO	0.25	0		0.25	0		0.07	0		0.06	0		0.09	0		0.02	0		0.02	0	
		YES	73.76	29.14	↑	849.90	9.57	↑	16.26	3.30	↑	20.17	3.04	↑	1.15	0.20	↑	6.81	1.12	↑	2.15	0.08	↑
		p-value	<0.01			<0.01			<0.01			<0.01			<0.01			<0.01			<0.01		
		Cliff's d	0.87 (L)			0.52 (L)			0.74 (L)			0.67 (L)			0.32 (S)			0.75 (L)			0.42 (M)		
	CDSP	NO	0.26	0		0.43	0		0.07	0		0.06	0		0.11	0		0.03	0		0.02	0	
		YES	9.36	2.10	↑	290.50	0	—	1.39	0	↑	0.57	0	↑	0.44	0	↑	0.94	0	↑	3.42	1.00	↑
		p-value	<0.01			0.3			0.04			0.02			<0.01			0.01			<0.01		
		Cliff's d	0.61 (L)			0.05 (N)			0.05 (N)			0.05 (N)			0.17 (S)			0.06 (N)			0.87 (L)		
	CC	NO	0.21	0		0.34	0		0.06	0		0.04	0		0.10	0		0.02	0		0.03	0	
		YES	63.00	12.60	↑	1573.00	46.81	↑	19.36	3.81	↑	15.68	1.93	↑	1.25	0.18	↑	9.29	1.40	↑	1.88	0.01	↑
		p-value	<0.01			<0.01			<0.01			<0.01			<0.01			<0.01			<0.01		
		Cliff's d	0.79 (L)			0.69 (L)			0.82 (L)			0.61 (L)			0.30 (S)			0.81 (L)			0.39 (M)		
	FD	NO	0.29	0		0.75	0		0.08	0		0.07	0		0.12	0		0.03	0		0.02	0	
		YES	-14.09	-4.00	↓	-5.59	-0.50	↓	-5.67	-1.37	↓	-4.50	-0.54	↓	-0.19	0	—	-2.60	-0.57	↓	-0.40	0	↓
		p-value	<0.01			<0.01			<0.01			<0.01			0.75			<0.01			<0.01		
		Cliff's d	-0.71 (L)			-0.51 (L)			-0.67 (L)			-0.49 (L)			0.01 (N)			-0.69 (L)			-0.22 (S)		
	SC	NO	0.17	0		1.02	0		0.04	0		-0.02	0		0.15	0		0.01	0		0.03	0	
		YES	134.00	36.29	↑	597.0	100.0	↑	15.09	6.34	↑	9.36	1.00	↑	1.27	0	—	5.84	3.00	↑	7.80	0.57	↑
		p-value	<0.01			<0.01			<0.01			<0.01			0.49			<0.01			<0.01		
		Cliff's d	0.93 (L)			0.52 (L)			0.66 (L)			0.52 (L)			0.06 (N)			0.59 (L)			0.67 (L)		

composition, the files affected by smells show a higher slope than clean files. This suggests that the files that will be affected by a smell exhibit a steeper growth in terms of metric values than files that are not becoming smelly. In other words, when a smell is going to appear, its operational indicators (metric value increases) occur very fast (not gradually). For example, considering the Apache ecosystem, we can see a clear difference between the growth of LOC in Blob and clean classes. Indeed, this latter have a mean growth in terms of LOC characterized by a slope of 0.40, while the slope for Blobs is, on average, 91.82. To make clear the interpretation of such data, let us suppose we plot both regression lines on the Cartesian plane. The regression line for Blobs will have an inclination of 89.38°, indicating an abrupt growth of LOC, while the inclination of the regression line for clean classes will be 21.8°, indicating less steep increase of LOC. The same happens when considering the LCOM cohesion metric (the higher the LCOM, the lower the class cohesion). For the overall dataset, the slope for classes that will become Blobs is 849.90 as compared to the 0.25 of clean classes. Thus, while the cohesion of classes generally decreases over time, classes destined to become Blobs exhibit cohesion metric loss orders of magnitude faster than clean classes. In general, the results in Table 3.4 show strong differences in the metrics' slope between clean and smelly files, indicating that it could be possible to create recommenders warning developers when the changes performed on a specific code component show a dangerous trend potentially leading to the introduction of a bad smell.

The Functional Decomposition (FD) smell deserves a separate discussion. As we can see in Table 3.4, the slope of the regression line for files affected by such a smell is negative. This means that during the evolution of files affected by Functional Decomposition we can observe a decrement (rather than an increment) of the metric values. The rationale behind such a result is intrinsic in the definition of this smell. Specifically, one of the symptoms of such a smell is represented by a class with a single action, such as a function. Thus, the changes that could introduce a Functional Decomposition might be the removal of responsibilities (*i.e.*, methods). This clearly results in the decrease of some metrics, such as NOM, LOC and WMC. As an example, let us consider the class `DisplayKMeans` of APACHE

Table 3.5: **RQ2**: Commit-goal tags to smell-introducing commits. BF: Bug Fixing; E: Enhancement; NF: New Feature; R: Refactoring. Results are reported in percentage.

Smell	Android				Apache				Eclipse				Overall			
	BF	E	NF	R	BF	E	NF	R	BF	E	NF	R	BF	E	NF	R
Blob	15	59	23	3	5	83	10	2	19	55	19	7	14	65	17	4
CDSP	11	52	30	7	6	63	30	1	14	64	18	4	10	60	26	4
CC	0	44	56	0	3	89	8	0	17	52	24	7	13	66	16	5
FD	8	48	39	5	16	67	14	3	18	52	24	6	16	60	20	4
SC	0	0	100	0	0	81	4	15	8	61	22	9	6	66	17	11

MAHOUT. The class implements the K-means clustering algorithm in its original form. However, after three commits the only operation performed by the class was the visualization of the clusters. Indeed, developers moved the actual implementation of the clustering algorithm in the class `Job` of the package `kmeans`, introducing a Functional Decomposition in `DisplayKMeans`.

Overall, by analyzing Table 3.4 we can conclude that (i) LOC characterizes the introduction of all the smells; (ii) LCOM, WMC, RFC and NOM characterize all the smells but Class Data Should be Private; (iii) CBO does not characterize the introduction of any smell; and (iv) the only metrics characterizing the introduction of Class Data Should be Private are LOC and NOA.

3.3.2 Why are code smells introduced?

To answer **RQ2**, we analyzed the percentage of smell-introducing commits classified according to the category of tags, *i.e., commit goal*, *project status*, and *developer status*.

Commit-Goal: Table 3.5 reports the percentage of smell-introducing commits assigned to each tag of the category *commit-goal*. Among the three different ecosystems analyzed, results show that smell instances are mainly introduced when developers perform enhancement operations on the system. When analyzing the

three ecosystems altogether, for all the considered types of smells the percentage of smell-introducing commits tagged as *enhancement* ranges between 60% and 66%. Note that by *enhancement* we mean changes applied by developers on existing features aimed at improving them. For example, a Functional Decomposition was introduced in the class `CreateProjectFromArchetypeMojo` of APACHE MAVEN when the developer performed the "*first pass at implementing the feature of being able to specify additional goals that can be run after the creation of a project from an archetype*" (as reported in the commit log).

Note that when considering *enhancement* or *new feature* all together, the percentage of smell-introducing commits exceeds, on average, 80%. This indicates, as expected, that the most smell-prone activities are performed by developers when adding new features or improving existing features. However, there is also a non-negligible number of smell-introducing commits tagged as *bug fixing* (between 6% and 16%). This means that also during corrective maintenance developers might introduce a smell, especially when the bug fixing is complex and requires changes to several code entities. For example, the class `SecuredModel` of APACHE JENA builds the security model when a semantic Web operation is requested by the user. In order to fix a bug that did not allow the user to perform a safe authentication, the developer had to update the model, implementing more security controls. This required changing several methods present in the class (10 out of 34). Such changes increase the whole complexity of the class (the WMC metric increased from 29 to 73) making `SecuredModel` a Complex Class.

Another interesting observation from the results reported in Table 3.5 is related to the number of smell-introducing commits tagged as *refactoring* (between 4% and 11%). While refactoring is the principal treatment to remove smells, we found 394 cases in which developers introduced new smells when performing refactoring operations. For example, the class `EC2ImageExtension` of APACHE JCLOUDS implements the `ImageExtension` interface, which provides the methods for creating an image. During the evolution, developers added methods for building a new image template as well as a method for managing image layout options (*e.g.*, its alignment) in the `EC2ImageExtension` class. Subsequently, a

developer performed an *Extract Class* refactoring operation aimed at reorganizing the responsibility of the class. Indeed, the developer split the original class into two new classes, *i.e.*, `ImageTemplateImpl` and `CreateImageOptions`. However, the developer also introduced a Functional Decomposition in the class `CreateImageOptions` since such a class, after the refactoring, contains just one method, *i.e.*, the one in charge of managing the image options. This result shows that refactoring can sometimes lead to unexpected side effects; besides the risk of introducing faults [124], when performing refactoring operations, there is also the risk of introducing design problems.

Table 3.6: **RQ2**: Project-Status tags to smell-introducing commits (in percentage).

Ecosystem	Smell	Working on Release				Project Startup			
		One Day	One Week	One Month	More	One Week	One Month	One Year	More
Android	Blob	7	54	35	4	6	3	35	56
	CDSP	14	20	62	4	7	17	33	43
	CC	0	6	94	0	0	12	65	23
	FD	1	29	59	11	0	4	71	25
	SC	0	0	100	0	0	0	0	100
Apache	Blob	19	37	43	1	3	7	54	36
	CDSP	10	41	46	3	3	8	45	44
	CC	12	30	57	1	2	14	46	38
	FD	5	14	74	7	3	8	43	46
	SC	21	18	58	3	3	7	15	75
Eclipse	Blob	19	37	43	1	3	20	32	45
	CDSP	10	41	46	3	6	12	39	43
	CC	12	30	57	1	2	12	42	44
	FD	5	14	73	8	2	5	35	58
	SC	21	18	58	3	1	5	19	75
Overall	Blob	15	33	50	2	5	14	38	43
	CDSP	10	29	58	3	6	12	39	43
	CC	18	28	53	1	4	13	42	41
	FD	7	22	66	5	3	7	42	48
	SC	16	20	58	6	2	6	17	75

Looking into the ecosystems, the general trend discussed so far holds for Apache and Eclipse. Regarding Android, we notice something different for Complex Class and Spaghetti Code smells. In these cases, the smell-introducing commits are mainly due to the introduction of new features. Such a difference could be due to the particular development model used for Android apps. Specifically, we manually analyzed the instances of smells identified in 70 Android apps, and we

observed that in the majority of the cases classes affected by a smell are those extending the Android `Activity` class, *i.e.*, a class extended by developers to provide features to the app's users. Specifically, we observed that quite often developers introduce a Complex Class or a Spaghetti Code smell when adding a new feature to their apps by extending the `Activity` class. For example, the class `ArticleViewActivity` of the AARD[10] app became a Complex Class after adding several new features (spread across 50 commits after its creation), such as the management of page buttons and online visualization of the article. All these changes contributed to increase the slope of the regression line for the RFC metric of a factor of 3.91 and for WMC of a factor of 2.78.

Table 3.7: **RQ2**: Developer-Status tags to smell-introducing commits (in percentage).

Ecosystem	Smell	Workload			Ownership		Newcomer	
		High	Medium	Low	True	False	True	False
Android	Blob	44	55	1	73	27	4	96
	CDSP	79	10	11	81	19	11	89
	CC	53	47	0	100	0	6	94
	FD	68	29	3	100	0	8	92
	SC	100	0	0	100	0	100	0
Apache	Blob	67	31	2	64	36	7	93
	CDSP	68	26	6	53	47	14	86
	CC	80	20	0	40	60	6	94
	FD	61	36	3	71	29	7	93
	SC	79	21	0	100	0	40	60
Eclipse	Blob	62	32	6	65	35	1	99
	CDSP	62	35	3	44	56	9	91
	CC	66	30	4	47	53	9	91
	FD	65	30	5	58	42	11	89
	SC	43	32	25	79	21	3	97
Overall	Blob	60	36	4	67	33	3	97
	CDSP	68	25	7	56	44	11	89
	CC	69	28	3	45	55	3	97
	FD	63	33	4	67	33	8	92
	SC	55	28	17	79	21	15	85

Project status: Table 3.6 reports the percentage of smell-introducing commits as-

[10] Aard is an offline Wikipedia reader.

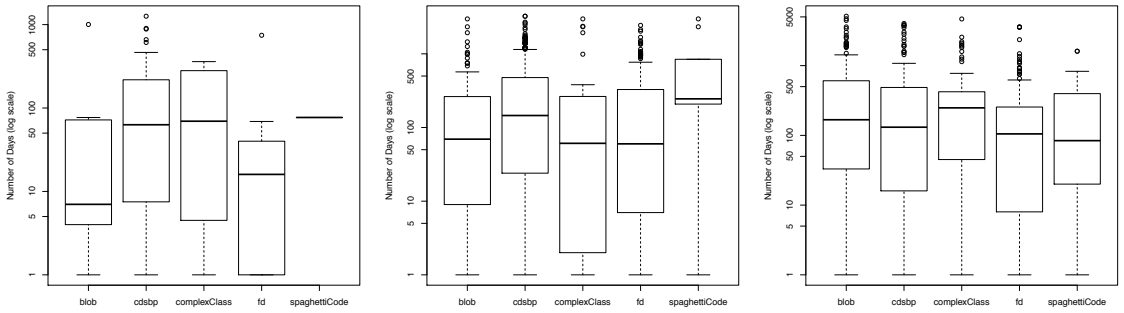
signed to each tag of the *project-status* category. As expected, most of the smells are introduced the last month before issuing a release. Indeed, the percentage of smells introduced more than one month prior to issuing a release is really low (ranging between 0% and 11%). This consideration holds for all the ecosystems and for all the bad smells analyzed, thus suggesting that the deadline pressure—assuming that release dates are planned —could be one of the main causes for smell introduction. Clearly, such a pressure might also be related to an expected more intense development activity (and a higher workload) developers are forced to bear to meet the deadline. Indeed, while we found no correlation in general between the distribution of commits and the distribution of code smell introduction (Spearman correlation value = -0.19), we observed a higher frequency of commits during the last month before a deadline, which tends to increase in the last week with a peak in the last day. This increasing rate of commits close to the deadline is also moderately correlated to a slightly increasing rate of code smell introduction during last month of activity and close to the deadline (Spearman correlation value = 0.516).

Considering the *project startup* tag, the results are quite unexpected. Indeed, a high number of smell instances are introduced few months after the project startup. This is particularly true for Blob, Class Data Should Be Private, and Complex Class, where more than half of the instances are introduced in the first year of systems' observed life history. Instead, Functional Decomposition, and especially Spaghetti Code, seem to be the types of smells that take more time to manifest themselves with more than 75% of Spaghetti Code instances introduced after the first year. This result contradicts, at least in part, the common wisdom that smells are introduced after several continuous maintenance activities and, thus, are more pertinent to advanced phases of the development process [8, 5].

Developer status: Finally, Table 3.7 reports the percentage of smell-introducing commits assigned to each tag of the *developer-status* category. From the analysis of the results it is evident that the developers' workload negatively influences the quality of the source code produced. On the overall dataset, at least in 55% of cases the developer who introduced the smell had a high workload. For example, on

the `InvokerMavenExecutor` class in APACHE MAVEN a developer introduced a Blob smell while adding the command line parsing to enable users to alternate the settings. When performing such a change, the developer had relatively high workload while working on nine other different classes (in this case, the workload was classified as high).

Developers who introduce smells are not newcomers, while often they are owners of smell-related files. This could look like an unexpected result, as the owner of the file—one of the most experienced developers of the file—is the one that has the higher likelihood of introducing a smell. However, it is clear that somebody who performs many commits has a higher chance of introducing smells. Also, as discussed by Zeller in his book *Why programs fail*, more experienced developers tend to perform more complex and critical tasks [151]. Thus, it is likely that their commits are more prone to introducing design problems.



(a) Android (b) Apache (c) Eclipse

Figure 3.3: Distribution of number of days a smell remained in the system before being removed.

3.3.3 What is the survivability of code smells?

We start by analyzing the data for smells that have been removed from the system, *i.e.*, those for which there is a closed interval delimited by a *last-smell-introducing commit* and *smell-removing-commit*. Figure 3.3 shows the box plot of the distribu-

tion of the number of days needed to fix a code smell instance for the different ecosystems. The box plots, depicted in log-scale, show that while few code smell instances are fixed after a long period of time (*i.e.*, even over 500 days) most of the instances are fixed in a relatively short time.

Table 3.8 shows the descriptive statistics of the distribution of the number of days when aggregating all code smell types considered in our study. We can notice considerable differences in the statistics for the three analyzed ecosystems. In particular, the median value of such distributions are 40, 101 and 135 days for Android, Apache and Eclipse projects, respectively.

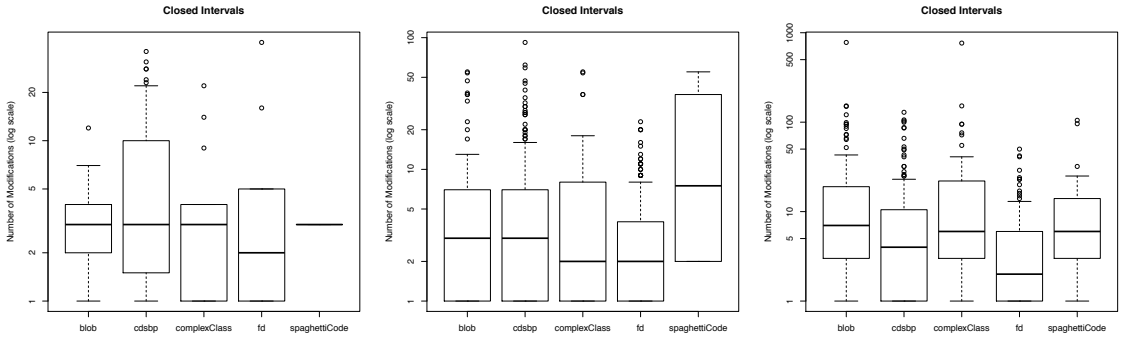
Table 3.8: Descriptive statistics of the number of days needed a smell remained in the system before being removed.

Ecosystem	Min	1st Qu.	Median	Mean	3rd Qu.	Max.
Android	0	5	40	140.8	196	1261
Apache	0	10	101	331.7	354	3244
Eclipse	0	21	135	435.2	446	5115

While it is difficult to speculate on the reasons why code smells are fixed quicker in the Android ecosystem than in the Apache and Eclipse ones, it is worth noting that on one hand Android apps generally have a much smaller size with respect to systems in the Apache and Eclipse ecosystems (*i.e.*, the average size, in terms of KLOC, is 415 for Android, while it is 1,417 for Apache and 1,534 for Eclipse), and on the other hand they have a shorter release cycles if compared with the other considered ecosystems. Because of these differences we decided to perform separate survivability analysis for the three ecosystems. As a consequence, we also selected a different threshold for each ecosystem when excluding code smell instances introduced too close to the end of the observed change history, needed to avoid cases in which the period of time needed for removing the smell is too short for being analyzed (see Section 3.2). Analyzing the distribution, we decided to choose the median as threshold, since it is a central value not affected by outliers, as opposed to the mean. Also, the median values of the distributions are small enough to consider discarded smells in the censored interval close to the end of

the observed change history (if compared for example to the mean time to remove a smell). Therefore, we used as threshold values 40, 101 and 135 days respectively for Android, Apache and Eclipse projects. Note that the censored intervals that we did not exclude were opportunely managed by the survival model.

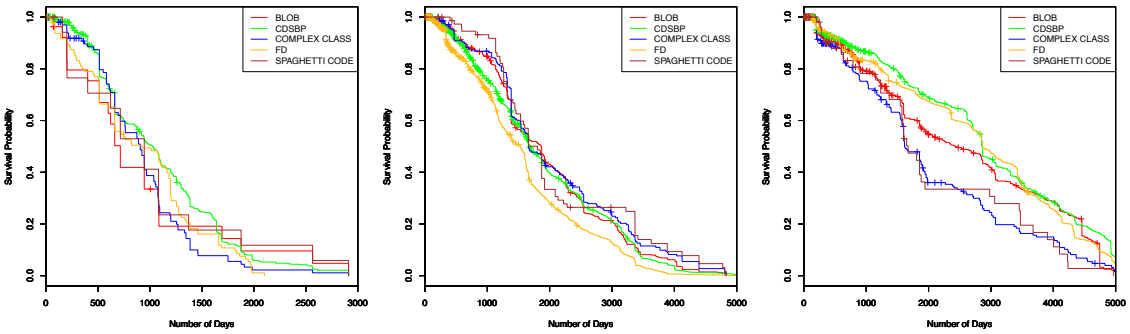
Figure 3.4 shows the number of modifications (*i.e.,* commits modifying the smelly file) performed by the developer between the introduction and the removal of the code smell instance. These results clearly show that most of the code smell instances are removed after a few commits, generally no more than five commits for Android and Apache, and ten for Eclipse. By combining what has been observed in terms of the number of days and the number of commits a smell remains in the system before being removed, we can conclude that if code smells are removed, this usually happens after few commits from their introduction, and in a relatively short time.



(a) Android (b) Apache (c) Eclipse

Figure 3.4: Distribution of number of commits between a smell introduction and its removal.

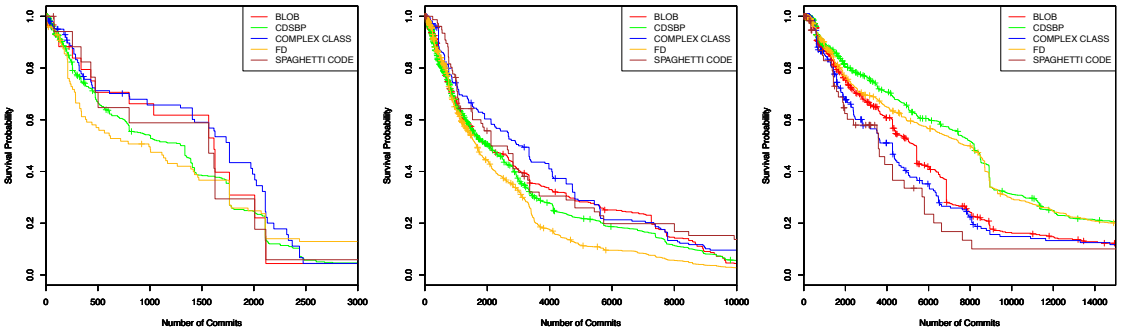
Figures 3.5 and 3.6 show the survivability curves for each type of code smell and for each ecosystem in terms of number of days and number of commits, respectively. Remember that, while the previous analysis was just limited to closed intervals (*i.e.,* smells that have been removed), here we also consider censored intervals (*i.e.,* smells that have been introduced but not removed until the last day of the analyzed change history). Overall, the plots show that the survivability of



(a) Android (b) Apache (c) Eclipse

Figure 3.5: Survival probability of code smells in terms of the number of days.

code smells is quite high. In particular, after 1,000 days, the survival probability of a code smell instance (*i.e.*, the probability that the code smell has not been removed yet) is around 50% for Android and 80% for Apache and Eclipse. Looking at the number of commits, after 2,000 commits the survival probability is still 30% for Android, 50% for Apache, and 75% for Eclipse.



(a) Android (b) Apache (c) Eclipse

Figure 3.6: Survival probability of code smells in terms of the number of commits.

These results may appear in contrast with respect to what has been previously observed while analyzing closed intervals. However, this is due to the very high percentage of unfixed code smells present in the subject systems and ignored in the closed intervals analysis. Table 3.9 provides an overview of the percentage of

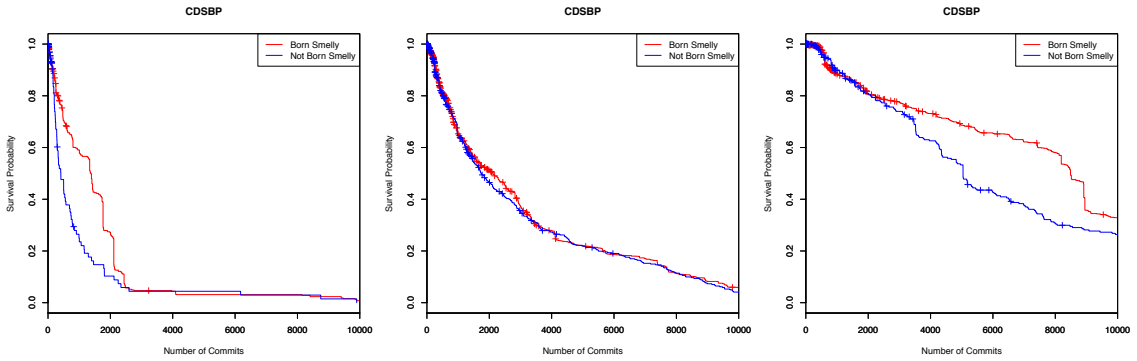
fixed and unfixed code smell instances found in the observable change history[11]. As we can see, the vast majority of code smells (81.4%, on average) are not removed, and this result is consistent across the three ecosystem (83% in Android, 87% in Apache, and 74% in Eclipse). The most refactored smell is the Blob with, on average, 27% of refactored instances. This might be due to the fact that such a smell is more visible than others due to the large size of the classes affected by it.

Table 3.9: Percentage of code smells removed and not in the observed change history.

Smell	Android		Apache		Eclipse	
	Removed	Not Removed	Removed	Not Removed	Removed	Not Removed
Blob	36	64	15	85	31	69
CDSBP	14	86	12	88	17	83
CC	15	85	14	86	30	70
FD	9	91	9	91	10	90
SC	11	89	13	87	43	57

Further insights about the survivability of the smells across the three ecosystems are provided in the survival models (*i.e.*, Figures 3.5 and 3.6). The survival of Complex Class (blue line) and Spaghetti Code (brown line) is much higher in systems belonging to the Apache ecosystem with respect to systems belonging to the Android and Eclipse ecosystems. Indeed, these two smell types are the ones exhibiting the highest survivability in Apache and the lowest survivability in Android and Eclipse. Similarly, we can notice that the survival curves for CDSBP (green) and FD (yellow) exhibit quite different shapes between Eclipse (higher survivability) and the other two ecosystems (lower survivability). Despite these differences, the outcome that can be drawn from the observation of the survival models is one and valid across all the ecosystems and for all smell types: *the survivability of code smells is very high, with over* 50% *of smell instances still "alive" after 1,000 days and 1,000 commits from their introduction*.

[11] As also done for the survival model, for the sake of consistency the data reported in Table 3.9 exclude code smell instances introduced too close to the end of the analyzed change history

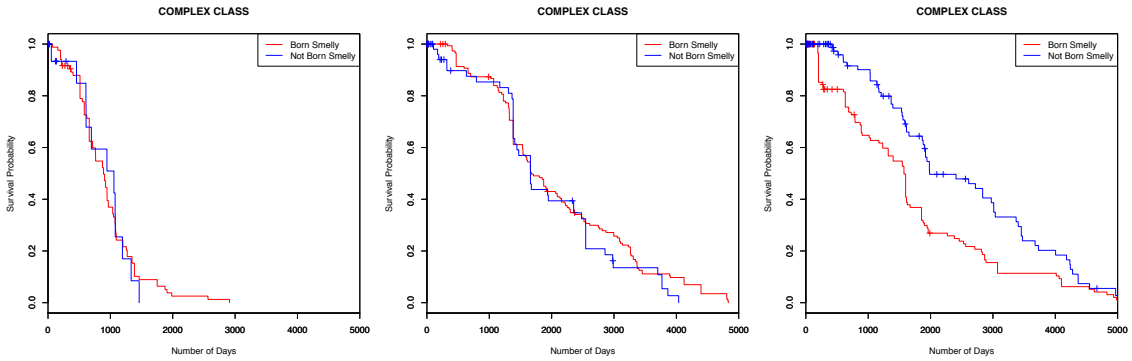


(a) Android (b) Apache (c) Eclipse

Figure 3.7: Survival probability of CDSBP instances affecting born and not born smelly artifacts.

Finally, we analyzed differences in the survivability of code smell instances affecting "born-smelly-artifacts" (*i.e.*, code files containing the smell instance since their creation) and "not-born-smelly-artifacts" (*i.e.*, code files in which the code smell has been introduced as a consequence of maintenance and evolution activities). Here there could be two possible scenarios: on the one hand developers might be less prone to refactor and fix born-smelly-artifacts than not-born-smelly-artifacts, since the code smell is somehow part of the original design of the code component. On the other hand, it could also be the case that the initial design is smelly because it is simpler to realize and release, while code smell removal is planned as a future activity. Both these conjectures have not been confirmed by the performed data analysis. As an example, we report the results achieved for the CDSBP and the Complex Class smell (the complete results are available in our online appendix [138]).

Figure 3.7 shows the survivability of born-smelly and not-born-smelly artifacts for the CDSBP instances. In this case, on two of the three analyzed ecosystems the survivability of born-smelly artifacts is actually higher, thus confirming in part the first scenario drawn above. However, when looking at the results for Complex Class instances (Figure 3.8), such a trend is not present in Android and Apache and it is exactly the opposite in Eclipse (*i.e.*, not-born-smelly-artifacts survive longer



(a) Android (b) Apache (c) Eclipse

Figure 3.8: Survival probability of Complex Class instances affecting born and not born smelly artifacts.

than the born-smelly ones). Such trends have also been observed for the other analyzed smells and, in some cases, contradictory trends were observed for the same smell in the three ecosystems (see [138]). Thus, it is not really possible to draw any conclusions on this point.

Table 3.10: How developers remove code smells.

Category	# Commits	% Percentage	% Excluding Unclear
Code Removal	329	34	40
Code Replacement	267	27	33
Unclear	158	16	-
Code Insertion	121	12	15
Refactoring	71	7	9
Major Restructuring	33	3	4

3.3.4 How do developers remove code smells?

Table 3.10 shows the results of the open coding procedure, aimed at identifying how developers fix code smells (or, more generally, how code smells are removed from the system). We defined the following categories:

- **Code Removal**. The code affected by the smell is deleted or commented. As a consequence, the code smell instance is no longer present in the system. Also, it is not replaced by other code in the smell-removing-commit.

- **Code Replacement**. The code affected by the smell is substantially rewritten. As a consequence, the code smell instance is no longer present in the system. Note that the code rewriting does not include any specific refactoring operation.

- **Code Insertion**. A code smell instance disappears after new code is added in the smelly artifact. While at a first glance it might seem unlikely that the insertion of new code can remove a code smell, the addition of a new method in a class could, for example, increase its cohesion, thus removing a Blob class instance.

- **Refactoring**. The code smell is explicitly removed by applying one or multiple refactoring operations.

- **Major Restructuring**. A code smell instance is removed after a significant restructuring of the system's architecture that totally changes several code artifacts, making it difficult to track the actual operation that removed the smell. Note that this category might implicitly include the ones listed above (*e.g.*, during the major restructuring some code has been replaced, some new code has been written, and some refactoring operations have been performed). However, it differs from the others since in this case we are not able to identify the exact code change leading to the smell removal. We only know that it is a consequence of a major system's restructuring.

- **Unclear**. The GitHub URL used to see the commit diff (*i.e.*, to inspect the changes implemented by the smell-removing-commit) was no longer available at the time of the manual inspection.

For each of the defined categories, Table 3.10 shows (i) the absolute number of smell-removing-commits classified in that category; (ii) their percentage over the

total of 979 instances and (iii) their percentage computed excluding the *Unclear* instances.

The first surprising result to highlight is that *only 9% (71) of smell instances are removed as a result of a refactoring operation*. Of these, 27 are Encapsulate Field refactorings performed to remove a CDSBP instance. Also, five additional CDSBP instances are removed by performing Extract Class refactoring. Thus, in these five cases the smell is not even actually fixed, but just moved from one class to another. Four Extract Class refactorings have been instead performed to remove four Blob instances. The Substitute Algorithm refactoring has been applied to remove Complex Classes (ten times) and Spaghetti code (four times). Other types of refactorings we observed (*e.g.,* move method, move field) were only represented by one or two instances. Note that this result (*i.e.,* few code smells are removed via refactoring operations) is in line with what was observed by Bazrfashan and Koschke [114] when studying how code clones had been removed by developers: They found that most of the clones were removed accidentally as a side effect of other changes rather than as the result of targeted code transformations.

One interesting example of code smell removed using an appropriate refactoring operation relates to the class `ConfigurationFactory` of the APACHE TOMEE project. The main responsibility of this class is to manage the data and configuration information for assembling an application server. Until the commit `0877b14`, the class also contained a set of methods to create new jars and descriptors for such jars (through the `EjbJar` and `EjbJarInfo` classes). In the commit mentioned above, the class affected by the Blob code smell has been refactored using Extract Class refactoring. In particular, the developer extracted two new classes from the original class, namely `OpenejbJar` and `EjbJarInfoBuilder` containing the extra functionalities previously contained in the original class.

The majority of code smell instances (40%) are simply removed due to the deletion of the affected code components. In particular: Blob, Complex Class, and Spaghetti Code instances are mostly fixed by removing/commenting large code fragments (*e.g.,* no longer needed in the system). In case of Class Data Should Be Private, the code smell frequently disappears after the deletion of public fields.

As an example of code smell removed via the deletion of code fragments, the class `org.apache.subversion.javahl.ISVNClient` of the APACHE SUBVERSION project was a Complex Class until the snapshot `673b5ee`. Then, the developers completely deleted several methods, as explained in the commit message: *"JavaHL: Remove a completely superfluous API"*. This resulted in the consequent removal of the Complex Class smell.

In 33% of the cases, smell instances are fixed by rewriting the source code in the smelly artifact. This frequently occurs in Complex Class and Spaghetti Code instances, in which the rewriting of method bodies can substantially simplify the code and/or make it more inline with object-oriented principles. Code Insertion represents 15% of the fixes. This happens particularly in Functional Decomposition instances, where the smelly artifacts acquire more responsibilities and are better shaped in an object-oriented flavor. Interestingly, also three Blob instances were removed by writing new code increasing their cohesion. An example of Functional Decomposition removed by adding code is represented by the class `ocl.library.executor.ExecutorFragment` of ECLIPSE OCL. The original goal of this class was to provide the description of the properties for the execution of the plug-in that allows users to parse and evaluate Object Constraint Language (OCL) constraints. In the commit `b9c93f8` the developers added to the class methods to access and modify such properties, as well as the `init` method, which provides APIs allowing external users to define their own properties. Finally, in 4% of the cases the smell instance was removed as a consequence of a major restructuring of the whole system.

3.4 Threats to Validity

The main threats related to the relationship between theory and observation (*construct validity*) are due to imprecisions/errors in the measurements we performed. Above all, we relied on *DECOR* rules to detect smells. Notice that our implementation uses the exact rules defined by Moha *et al.* [20], and has been already used in other our previous work [52]. Nevertheless, we are aware that our results can

be affected by (i) the thresholds used for detecting code smell instances, and (ii) the presence of false positives and false negatives.

A considerable increment/decrement of the thresholds used in the detection rules might determine changes in the set of detected code smells (and thus, in our results). In our study we used the thresholds suggested in the paper by Moha *et al.* [20]. As for the presence of false positives and false negatives, Moha *et al.* reported for *DECOR* a precision above 60% and a recall of 100% on Xerces 2.7.0. As for the precision, other than relying on Moha *et al.* assessment, we have manually validated a subset of the 4,627 detected smell instances. This manual validation has been performed by two authors independently[12], and cases of disagreement were discussed. In total, 1,107 smells were validated, including 241 Blob instances, 317 Class Data Should Be Private, 166 Complex Class, 65 Spaghetti Code, and 318 Functional Decomposition. Such a (stratified) sample is deemed to be statistically significant for a 95% confidence level and $\pm 10\%$ confidence interval [152]. The results of the manual validation indicated a mean precision of 73%, and specifically 79% for Blob, 62% for Class Data Should Be Private, 74% for Complex Class, 82% for Spaghetti Code, and 70% for Functional Decomposition. In addition, we replicated all the analysis performed to answer our research questions by just considering the smell-introducing commits (2,555) involving smell instances that have been manually validated as true positives. The results achieved in this analysis (available in our replication package [138]) are perfectly consistent with those obtained in our work on the complete dataset, thus confirming all our findings. Finally, we are aware that our study can also suffer from the presence of false negatives. However, (i) the sample of investigated smell instances is pretty large (4,627 instances), and (ii) the *DECOR*'s claimed recall is very high.

Another threat related to the use of *DECOR* is the possible presence of "conceptual" false positive instances [11], *i.e.,* instances detected by the tool as true positives but irrelevant for developers. However, most of the code smells studied in this chapter (*i.e., Blob*, *Complex Class* and *Spaghetti Code*) have been shown to be perceived as harmful by developers [50]. This limits the possible impact of this

[12]The thesis' author was equally involved in this task.

Table 3.11: Metrics used by the detector compared to the metrics evaluated in **RQ1**.

Code Smell	Metrics used by DECOR	Metrics used in **RQ1**	Overlap
Blob	#Methods*, #Attributes* LCOM*, MethodName, ClassName	LOC, LCOM*, WMC, RFC, CBO #Methods*, #Attributes*	3 metrics out of 5 used by DECOR. Note that in this case DECOR also uses textual aspects of the source code that we do not take into account in the context of RQ1.
CDSBP	# Public Attributes	LOC, LCOM, WMC, RFC, CBO #Methods, #Attributes	–
Complex Class	WMC	LOC, LCOM, WMC* RFC, CBO #Methods, #Attributes	1 metric in overlap between the two sets. Note that in the chapter we did not only observe the growth of the WMC metric, but we found that other several metrics tend to increase over time for the classes that will become smelly (e.g., LCOM and NOA).
Functional Decomposition	# Private Attributes, #Attributes* Class name	LOC, LCOM, WMC, RFC, CBO #Methods, #Attributes*	1 metric in overlap. Also in this case, we found decreasing trends for all the metrics used in RQ1, and not only for the one used by DECOR.
Spaghetti Code	Method LOC, #Parameters DIT	LOC, LCOM, WMC, RFC, CBO #Methods, #Attributes	–

threat.

The overlap between the quality metrics used when building the linear regression models (**RQ1**) and the metrics used by DECOR for detecting code smells may bias the findings related to when code smells are introduced. In our empirical investigation we are not interested in predicting the presence of code smells over time, but we want to observe whether the trends of quality metrics are different for classes that will become smelly with respect to those that will not become smelly. For this reason, the use of indicators that are used by the detector to identify smells should not influence our observations. However, in most of the cases we avoided the overlap between the metrics used by DECOR and the ones used in the context of **RQ1**. Table 3.11 reports, for each smell, (i) the set of metrics used by the detector, (ii) the set of metrics evaluated in the context of **RQ1**, and (iii) the overlap between them. We can note that the overlap between the two sets of metrics is often minimal or even empty (*e.g.*, in the case of *Spaghetti Code*). Also, it is worth noting that the detector uses specific thresholds for detecting smells, while in our case we simply look for the changes of metrics' value over time.

As explained in Section 3.2, the heuristic for excluding projects with incomplete history from the *Project startup* analysis may have failed to discard some projects. Also, we excluded the first commit from a project's history involving Java files from the analysis of smell-introducing commits, because such commits are likely to be imports from old versioning systems, and, therefore, we only focused our attention (in terms of the first commit) on the addition of new files during the observed history period. Concerning the tags used to characterize smell-introducing

changes, the commit classification was performed by two different authors and results were compared and discussed in cases of inconsistencies. Also, a second check was performed for those commits linked to issues (only 471 out of 9,164 commits), to avoid problems due to incorrect issue classification [153, 154].

The analysis of developer-related tags was performed using the *Git author* information instead of relying on *committers* (not all authors have commit privileges in open source projects, hence observing committers would give an imprecise and partial view of the reality). However, there is no guarantee that the reported authorship is always accurate and complete. We are aware that the *Workload* tag measures the developers' activity within a single project, while in principle one could be busy on other projects or different other activities. One possibility to mitigate such a threat could have been to measure the workload of a developer within the entire ecosystem. However, in our opinion, this would have introduced some bias, *i.e.,* assigning a high workload to developers working on several projects of the same ecosystem and a low workload to those that, while not working on other projects of the same ecosystem, could have been busy on projects outside the ecosystem. It is also important to point out that, in terms of the relationship between *Workload* tag and smell introduction, we obtained consistent results across three ecosystems, which at least mitigates the presence of a possible threat. Also, estimating the *Workload* by just counting commits is an approximation. However, we do not use the commit size because there might be a small commit requiring a substantial effort as well.

As for **RQ3**, the heuristic used to assess when a smell has been introduced and when a smell disappears from the system is to be considered as conservative. The starting point of the history of a smell is the *last-smell-introducing-commit*, *i.e.,* the final commit leading to the introduction of a smell. As commit removing the smell, we identified the *smell-removing commit*, *i.e.,* the commit in which DECOR did not identify the smell anymore is considered. While other commits that are subsequent to the *smell-removing commit* may possibly contribute to the decreasing of the smelliness of a code element, the selected ending point should be considered as the one causing the decisive reduction of the smelliness (indeed, after that

commit *DECOR* is not able to identify the component as smelly).

The proxies that we used for the survivability of code smells (*i.e.*, the number of days and the number of commits from their introduction to their removal) should provide two different views on the survivability phenomenon. However, the level of activity of a project (*e.g.*, the number of commits per week) may substantially change during its lifetime, thus, influencing the two measured variables.

When studying the survival and the time to fix code smell instances, we relied on DECOR to assess when a code smell instance has been fixed. Since we rely on a metric-based approach, code smell instances whose metrics' values alternate between slightly below and slightly above the detection threshold used by DECOR appear as a series of different code smell instances having a short lifetime, thus introducing imprecisions in our data. To assess the extent of such imprecisions, we computed the distribution of a number of fixes for each code file and each type of smell in our dataset. We found that only between 0.7% and 2.7% (depending on the software ecosystem) of the files has been fixed more than once for the same type of code smell during the considered change history. Thus, such a phenomenon should only marginally impact our data.

Concerning **RQ4**, we relied on an open coding procedure performed on a statistically significant sample of *smell-removing commits* in order to understand how code smells are removed from software systems. This procedure involved three of the authors and included open discussion aimed at double checking the classifications individually performed. Still, we cannot exclude imprecision and some degree of subjectiveness (mitigated by the discussion) in the assignment of the *smell-removing commits* to the different fixing/removal categories.

As for the threats that could have influenced the results (*internal validity*), we performed the study by comparing classes affected (and not) by a specific type of smell. However, there can also be cases of classes affected by different types of smells at the same time. Our investigation revealed that such classes represent a minority (3% for Android, 5% for Apache, and 9% for Eclipse), and, therefore, the coexistence of different types of smells in the same class is not particularly interesting to investigate, given also the complexity it would have added to the

Table 3.12: Number of censored intervals discarded using different thresholds. Percentages are reported between brackets.

# Censored Intervals	Android	Apache	Eclipse
Total	708	5780	2709
Discarded using 1st Q.	1 (0.1)	43 (0.7)	7 (0.3)
Discarded using Median	3 (0.4)	203 (3.5)	51 (1.9)
Discarded using 3rd Q.	26 (3.7)	602 (10.0)	274 (10.0)

study design and to its presentation. Another threat could be represented by the fact that a commit identified as a smell-removing-commit (*i.e.,* a commit which fixes a code smell) could potentially introduce another type of smell in the same class. To assess the extent to which this could represent a threat to our study, we analyzed in how many cases this happened in our entire dataset. We found that in only four cases a fix of a code smell led to the introduction of a different code smell type in the same software artifact.

In **RQ2** we studied tags related to different aspects of a software project's lifetime, characterizing commits, developers, and the project's status itself, but we are aware that there could be many other factors that could have influenced the introduction of smells. In any case, it is worth noting that it is beyond the scope of this work to make any claims related to causation of the relationship between the introduction of smells and product or process factors characterizing a software project.

The survival analysis in the context of **RQ3** has been performed by excluding smell instances for which the developers had not "enough time" to fix them, and in particular censored intervals having the *last-smell-introducing commit* too close to the last commit analyzed in the project's history. Table 3.12 shows the absolute number of censored intervals discarded using different thresholds. In our analysis, we used the median of the smelly interval (in terms of the number of days) for closed intervals as a threshold. As we can observe in Table 3.12, this threshold allows the removal of a relatively small number of code smells from the analysis. Indeed, we discarded 3 instances (0.4% of the total number of censored intervals)

Table 3.13: Descriptive statistics of the number of days of censored intervals.

Ecosystem	Min	1st Qu.	Median	Mean	3rd Qu.	Max.
Android	3	513	945	1,026	1,386	2,911
Apache	0	909	1,570	1,706	2,434	5,697
Eclipse	0	1,321	2,799	2,629	4,005	5,151

in Android, 203 instances (3.5%) in Apache and 51 instances (1.9%) in Eclipse.

This is also confirmed by the analysis of the distribution of the number of days composing the censored intervals, shown in Table 3.13, which highlights how the number of days composing censored intervals is quite large. It is worth noting that if we had selected the first quartile as threshold, we would have removed too few code smells from the analysis (*i.e.,* 1 instance in Android, 43 in Apache, and 7 in Eclipse). On the other hand, a more conservative approach would have been to exclude censored data where the time interval between the *last-smell-introducing commit* and the last analyzed commit is greater than the third quartile of the smell removing time distribution. In this case, we would have removed a higher number of instances with respect to the median (*i.e.,* 26 instances in Android, 602 in Apache, and 51 in Eclipse). Moreover, as we show in our online appendix [138], this choice would have not impacted our findings (*i.e.,* the achieved results are consistent with what we observed by using the median). Finally, we also analyzed the proportion of closed and censored intervals considering (i) the original change history (no instance removed), (ii) the first quartile as threshold, (iii) the median value as threshold, and (iv) the third quartile as threshold. As shown in our online appendix [138], we found that the proportion of closed and censored intervals after excluding censored intervals using the median value, remains almost identical to the initial proportion (*i.e.,* original change history). Indeed, in most of the cases the differences is less than 1%, while in only few cases it reaches 2%.

Still in the context of **RQ3**, we considered a code smell as removed from the system in a commit c_i when DECOR detects it in c_{i-1} but does not detect it in c_i. This might lead to some imprecisions why computing the lifetime of the smells. Indeed, suppose that a file f was affected by the Blob smell until commit c_i (*i.e.,*

DECOR still identify f as a Blob class in commit c_i). Then, suppose that f is completely rewritten in c_{i+1} and that DECOR still identifies f as a Blob class. While it is clear that the Blob instance detected in commit c_i is different with respect to the one detected in commit c_{i+1} (since f has been completely rewritten), we are not able to discriminate the two instances since we simply observe that DECOR was detecting a Blob in f at commit c_i and it is still detecting a Blob in f at commit c_{i+1}. This means that (i) we will consider for the Blob instance detected at commit c_i a lifetime longer than it should be, and (ii) we will not be able to study a new Blob instance. Also, when computing the survivability of the code smells we considered the smell introduced only after the *last-smell-introducing-commit* (*i.e.*, we ignored the other commits contributing to the introduction of the smell). Basically, our **RQ3** results are conservative in the sense that they consider the minimum survival time of each studied code smell instance.

The main threats related to the relationship between the treatment and the outcome (*conclusion validity*) are represented by the analysis method exploited in our study. In **RQ1**, we used non-parametric tests (Mann-Whitney) and effect size measures (Cliff's Delta), as well as regression analysis. Results of **RQ2** and **RQ4** are, instead, reported in terms of descriptive statistics and analyzed from purely observational point of view. As for **RQ3**, we used the Kaplan-Meier estimator [148], which estimates the underlying survival model without making any initial assumption upon the underlying distribution.

Finally, regarding the generalization of our findings (*external validity*) this is, to the best of our knowledge, the largest study—in terms of the number of projects (200)—concerning the analysis of code smells and of their evolution. However, we are aware that we limited our attention to only five types of smells. As explained in Section 3.2, this choice is justified by the need for limiting the computational time since we wanted to analyze a large number of projects. Also, we tried to diversify the types of smells by including smells representing violations of OO principles and "size-related" smells. Last, but not least, we made sure to include smells—such as Complex Class, Blob, and Spaghetti Code—that previous studies indicated to be perceived by developers as severe problems [50]. Our choice of the

subject systems is not random, but guided by specific requirements of our underlying infrastructure. Specifically, the selected systems are written in Java, since the code smell detector used in our experiments is able to work with software systems written in this programming language. Clearly, results cannot be generalized to other programming languages. Nevertheless, further studies aiming at replicating our work on other smells, with projects developed for other ecosystems and in other programming languages, are desirable.

3.5 Conclusion

This chapter presented a large-scale empirical study conducted over the commit history of 200 open source projects and aimed at understanding *when* and *why* bad code smells are introduced, *what* is their survaivability, and under which circumstances they are removed. These results provide several valuable findings for the research community:

Lesson 1. *Most of the times code artifacts are affected by bad smells since their creation*. This result contradicts the common wisdom that bad smells are generally introduced due to several modifications made on a code artifact. Also, this finding highlights that the introduction of most smells can simply be avoided by performing quality checks at commit time. In other words, instead of running smell detectors time-to-time on the entire system, these tools could be used during commit activities (in particular circumstances, such as before issuing a release) to avoid or, at least, limit the introduction of bad code smells.

Lesson 2. *Code artifacts becoming smelly as consequence of maintenance and evolution activities are characterized by peculiar metrics' trends, different from those of clean artifacts*. This is in agreement with previous findings on the historical evolution of code smells [15, 52, 12]. Also, such results encourage the development of recommenders able to alert software developers when changes applied to code artifacts result in worrisome metric trends, generally characterizing artifacts that will be affected by a smell.

Lesson 3. *While implementing new features and enhancing existing ones, the main activities during which developers tend to introduce smells, we found almost 400 cases in which refactoring operations introduced smells*. This result is quite surprising, given that one of the goals behind refactoring is the removal of bad smells [8]. This finding highlights the need for techniques and tools aimed at assessing the impact of refactoring operations on source code before their actual application (*e.g.*, see the recent work by Chaparro *et al.* [155]).

Lesson 4. *Newcomers are not necessary responsible for introducing bad smalls, while developers with high workloads and release pressure are more prone to introducing smell instances*. This result highlights that code inspection practices should be strengthened when developers are working under these stressful conditions.

Lesson 5. *Code Smells have a high survivability and are rarely removed as a direct consequence of refactoring activities*. We found that 80% of the analyzed code smell instances survive in the system and only a very low percentage of them (9%) is removed through the application of specific refactorings. While we cannot conjecture on the reasons behind such a finding (*e.g.*, the absence of proper refactoring tools, the developers' perception of code smells, *etc.*), our results highlight the need for further studies aimed at understanding why code smells are not refactored by developers. Only in this way it will be possible to understand where the research community should invest its efforts (*e.g.*, in the creation of a new generation of refactoring tools).

These lessons learned represent the main input for our future research agenda on the topic, mainly focusing on designing and developing a new generation of code quality-checkers, such as those described in Lesson 2, as well as investigating the reasons behind developers' lack of motivation to perform refactoring activities, and which factors (*e.g.*, intensity of the code smell) promote/discourage developers to fix a smell instance (Lesson 5). Also, we intend to perform a deeper investigation of factors that can potentially explain the introduction of code smells, other than the ones already analyzed in this chapter.

Chapter 4

On the Diffuseness and the Impact on Maintainability of Code Smells: A Large Scale Empirical Investigation

4.1 Introduction

Bad code smells (shortly "code smells" or "smells") have been defined by Fowler as symptoms of poor design and implementation choices applied by programmers during the development of a software project [8]. As a form of technical debt [4], they could hinder the comprehensibility and maintainability of software systems [131]. An example of code smell is the *God Class*, a large and complex class that centralizes the behavior of a portion of a system and only uses other classes as data holders. *God Classes* can rapidly grow out of control, making it harder and harder for developers to understand them, to fix bugs, and to add new features.

Code smells have been studied by the research community from different perspectives. On the one side, researchers have developed methods and tools to detect code smells. Such tools exploit different kinds of approaches, for example approaches using constraints upon metrics [20, 21, 22, 26], graph-based approaches [25], historical analysis of source code changes [52], or search-based optimization techniques in which rules are generated to maximize the detection of some known

or artificial smells [32, 35].

On the other side, researchers have investigated how relevant code smells are for developers [9, 50], when and why they are introduced [47], their evolution over time [14, 13, 15, 12], and their impact on software quality properties, such as program comprehensibility [18], fault- and change-proneness [17, 16, 41], maintenance effort [19], and, in general, maintainability [10, 9, 119, 42, 43].

Similarly to some previous work [17, 42, 43, 44] this study focuses on the relationship existing between the occurrence of code smells in software projects and the software change- and fault-proneness. Specifically, while previous work suggested a significant correlation between smells and change and fault-proneness, the evidence currently available is limited and suggests the need for further studies because of:

- **Limited size of previous studies:** the study of Khomh *et al.* [17] has been conducted on four open source systems, while the study by D'Ambros *et al.* [41] has been performed on 7 systems. Still, the studies by Li and Shatnawi [42], Olbrich *et al.* [43], and Gatrell and Counsell [44] have been conducted considering the change history of only one software project.

- **Detected smells vs. manually validated smells:** Previous work which analyze the impact of code smells on change- and fault-proneness, including the one by Khomh *et al.* [17], relied on data obtained from smell detectors. Although such smell detector are often able to achieve a good level of accuracy it is still possible that their intrinsic imprecision affects the study results.

- **Lack of analysis of the observed phenomenon's magnitude:** while previous work indicated how some smells can be more harmful than others, such an analysis did not take into account the magnitude of the observed phenomenon. For example, even if a smell type results to be very harmful, this may or may not be relevant depending on the diffuseness of such a smell in software projects;

- **Lack of analysis of the effect's magnitude:** Previous work indicated that,

if a class is affected by a code smell, it has more chances to exhibit defects (or to undergo changes) than other classes. However, no study has observed the magnitude of such changes and defects, *i.e.,* by addressing the question: How many defects would exhibit on average a class affected by a code smell compared with another class affected by a different type of smell, or not affected by any smell?

- **Lack of within-artifact analysis:** sometimes, a class has intrinsically a very high change-proneness and even a very high fault-proneness, *e.g.,* because it plays a core role in the system, or because implements a very complex feature. Hence, the class may be intrinsically "smelly", and developers are aware of that [50, 47]. Instead, there may be classes that becomes smelly during their lifetime [47], because of maintenance activities. Or else, classes where the smell has been removed, possibly because of refactoring activities [49]. For such classes, it is of paramount importance to compare the change- and fault-proneness when the class contained a smell and when not, in order to better relate the cause (presence of smell) with the possible effect (change- or fault-proneness).

- **Lack of a temporal relation analysis between smell presence and fault introduction:** While previous work correlated the presence of code smells with high fault- and change-proneness, one may wonder whether the artifact was smelly when the fault was introduced, or whether the fault was introduced before the class become smelly.

To cope with the aforementioned issues, this chapter aims at corroborating previous empirical research on the impact and effect of code smells in software projects, by analyzing their diffuseness and effect on change- and fault-proneness on a large set of software projects. In the context of this chapter, the "diffuseness" of a code smell type (*e.g.,* God Class) refers to the percentage of code components in a system affected by at least one of its instances. The study has been conducted on a total of 395 releases of 30 open source systems, and considered 13 different types of code smells. More specifically, the study aims at investigating:

1. *What is the diffuseness of code smells in open source systems*. If the magnitude of the phenomenon is small—*i.e.*, code smells, or some specific types of code smells, are poorly diffused—then effort spent in identifying and refactoring them might not be worthwhile.

2. *What is the impact of code smells on maintenance properties, and specifically on code change- and fault-proneness*. We intend to investigate toort what extent the previous findings reported by Khomh *et al.* [17] and D'Ambros *et al.* [41]—obtained on a small set of software systems and based on smells automatically identified using code smell detectors—are confirmed on a larger set of 395 software releases, and considering manually validated smell instances.

To the best of our knowledge, this is to date the largest study investigating the relationship between the presence of code smells and the source code change- and fault-proneness. In addition, and to cope with the other limitations of previous studies mentioned above, this chapter (i) relies on a set of manually-validated code smells rather than just on the output of tools, (ii) analyzes the fault proneness magnitude in terms of number of code smells, (iii) performs an analysis of the evolution of classes in order to investigate how the change/fault-proneness changes when the smell has been removed, and (iv) uses the SZZ algorithm [156] to determine whether an artifact was already smelly when a fault was induced. The dataset used in this study is publicly available in our online appendix [157].

4.2 Study Definition and Planning

The *goal* of this study is to analyze the diffuseness of 13 code smells in real software applications and their impact on code change- and fault-proneness. It is worth remarking that the term "diffuseness", when associated to a type of code smells, refers to the percentage of code components in a system affected by at least one of its instances. Analyzing the diffuseness of code smells is a preliminary analysis needed to better interpret the smells' effect on change- and fault-proneness.

Indeed, some smells might be highly correlated with fault-proneness but rarely diffused in software projects or *vice versa*. The 13 considered code smells are reported in Table 4.1 together with a short description.

4.2.1 Research Questions and Planning

In the context of our study we formulated the following three research questions:

- **RQ1**: *What is the diffuseness of code smells in software systems?* This research question is preliminary research question aiming at assessing to what extent software systems are affected by code smells.
- **RQ2**: *To what extent do classes affected by code smells exhibit a different level of change- and fault-proneness with respect to classes do not affected by code smells?* Previous work [17] found that classes participating in at least one smell have a higher chance of being change- and fault-prone than other classes. In this work we are interested in measuring the change- and fault-proneness magnitude of such classes, in terms of number of changes and of bug fixes.
- **RQ3**: *To what extent does the change- and fault-proneness of classes vary when code smells are introduced and when they are removed?* This research question investigates whether the change- and fault-proneness of a class increases when a smell has been introduced, and whether it decreases when the smell has been refactored. Such an analysis is of paramount importance because a class may be intrinsically change-prone (and also fault-prone) regardless of whether it exhibits code smells or not.

To answer our research questions we mined 395 releases of 30 open source systems searching for instances of the 13 code smells object of our study. Table 4.2 reports the analyzed systems, the number of releases considered for each of them, and their size ranges in terms of number of classes, number of methods, and KLOCs. The choice of the systems to consider was not random but guided by the will to consider object systems having different size (ranging from 0.4 to 868 KLOCs), belonging to different application domains (modeling tools, parsers,

Table 4.1: The Code Smells considered in our Study

Name	Description
Class Data Should Be Private (CDSBP)	A class exposing its fields, violating the principle of data hiding.
Complex Class	A class having at least one method having a high cyclomatic complexity.
Feature Envy	A method is more interested in a class other than the one it actually is in.
God Class	A large class implementing different responsibilities and centralizing most of the system processing.
Inappropriate Intimacy	Two classes exhibiting a very high coupling between them.
Lazy Class	A class having very small dimension, few methods and low complexity.
Long Method	A method that is unduly long in terms of lines of code.
Long Parameter List (LPL)	A method having a long list of parameters, some of which avoidable.
Message Chain	A long chain of method invocations is performed to implement a class functionality.
Middle Man	A class delegates to other classes most of the methods it implements.
Refused Bequest	A class redefining most of the inherited methods, thus signaling a wrong hierarchy.
Spaghetti Code	A class implementing complex methods interacting between them, with no parameters, using global variables.
Speculative Generality	A class declared as abstract having very few children classes using its methods.

Table 4.2: Systems involved in the study

System	#Releases	Classes	Methods	KLOCs
ArgoUML	16	777-1,415	6,618-10,450	147-249
Ant	22	83-813	769-8,540	20-204
aTunes	31	141-655	1,175-5,109	20-106
Cassandra	13	305-586	1,857-5,730	70-111
Derby	9	1,440-1,929	20,517-28,119	558-734
Eclipse Core	29	744-1,181	9,006-18,234	167-441
Elastic Search	8	1,651-2,265	10,944-17,095	192-316
FreeMind	16	25-509	341-4,499	4-103
Hadoop	9	129-278	1,089-2,595	23-57
HSQLDB	17	54-444	876-8,808	26-260
Hbase	8	160-699	1,523-8148	49-271
Hibernate	11	5-5	15-18	0.4-0.5
Hive	8	407-1,115	3,725-9,572	64-204
Incubating	6	249-317	2,529-3,312	117-136
Ivy	11	278-349	2,816-3,775	43-58
Lucene	6	1,762-2,246	13,487-17,021	333-466
JEdit	23	228-520	1,073-5,411	39-166
JHotDraw	16	159-679	1,473-6,687	18-135
JFreeChart	23	86-775	703-8,746	15-231
JBoss	18	2,313-4,809	19,901-37,835	434-868
JVlt	15	164-221	1,358-1,714	18-29
jSL	15	5-10	26-43	0.5-1
Karaf	5	247-470	1,371-2,678	30-56
Nutch	7	183-259	1,131-1,937	33-51
Pig	8	258-922	1,755-7,619	34-184
Qpid	5	966-922	9,048-9,777	89-193
Sax	6	19-38	119-374	3-11
Struts	7	619-1,002	4,059-7,506	69-152
Wicket	9	794-825	6,693-6,900	174-179
Xerces	16	162-736	1,790-7,342	62-201
Total	**395**	**5-4,809**	**15-37,835**	**0.4-868**

IDEs, IR-engines, etc), developed by different open source communities (`Apache`, `Eclipse`, *etc.*), and having different lifetime (from 1 to 19 years).

Table 4.3: The Rules used by our Tool to Detect Candidate Code Smells

Name	Description
CDSBP	A class having at least one public field.
Complex Class	A class having at least one method for which McCabe cyclomatic complexity is higher than 10.
Feature Envy	All methods having more calls with another class than the one they are implemented in.
God Class	All classes having (i) cohesion lower than the average of the system AND (ii) LOCs > 500.
Inappropriate Intimacy	All pairs of classes having a number of method's calls between them higher than the average number of calls between all pairs of classes.
Lazy Class	All classes having LOCs lower than the first quartile of the distribution of LOCs for all system's classes.
Long Method	All methods having LOCs higher than the average of the system.
LPL	All methods having a number of parameters higher than the average of the system.
Message Chain	All chains of methods' calls longer than three.
Middle Man	All classes delegating more than half of the implemented methods.
Refused Bequest	All classes overriding more than half of the methods inherited by a superclass.
Spaghetti Code	A class implementing at least two long methods interacting between them through method calls or shared fields.
Speculative Generality	A class declared as abstract having less than three children classes using its methods.

The need for analyzing smells in 395 project releases makes the manual detection of the 13 code smells prohibitively expensive. For this reason, we developed a simple tool to perform smell detection. The tool outputs a list of candidate code components (*i.e.*, classes or methods) potentially exhibiting a smell. Then, we manually validated the candidate code smells suggested by the tool. The validation was performed by two of the authors[1] who individually analyzed and classified as true positive or false positive all candidate code smells. Finally, they performed an open discussion to resolve possible conflicts and reach a consensus on the detected code smells.

To ensure high recall, our detection tool uses very simple rules that overesti-

[1]The thesis' author was equally involved in this task.

mate the presence of code smells. Such rules, reported in Table 4.3, are inspired to the rule cards proposed by Moha *et al.* [20] in *DECOR*, *i.e.,* our tool considers the metric profile of classes/methods.

The metrics' thresholds (needed for discriminating whether a class/method is affected or not by a smell) have been set lowering the thresholds used by Moha *et al.* [20]. Again, this was done in order to detect as many code smell instances as possible. For example, in the case of the *Complex Class* smell, we select as candidates all classes having a cyclomatic complexity higher than 10. Such a choice was driven by recent findings reported by Lopez *et al.* [158], which found that *"a threshold lower than 10 is not significant in Object-Oriented programming when interpreting the complexity of a method"*. As for the other smells, we relied on (i) simple filters, *e.g.,* in the cases of *CDSBP* (where we discarded from the manual validation all the classes having no public attributes) and *Feature Envy* (we only consider the methods having more relationships toward another class than with the class they are contained in), (ii) the analysis of the metrics' distribution (like in the cases of *Lazy Class*, *Inappropriate Intimacy*, *Long Method*, and *Long Parameter List*), or (iii) very conservative thresholds (*e.g.,* a God Class should not have less than 500 LOCs).

Note that we chose not to use existing detection tools [21, 23, 35, 25, 20, 24, 52] because (i) none of them has ever been applied to detect all the studied code smells and (ii) their detection rules are generally more restrictive to ensure a good compromise between recall and precision and thus may miss some smell instances. To verify this claim, we evaluated the behavior of three existing tools, *i.e.,* DECOR [20], JDeodorant [25], and HIST [52] on one of the systems used in the empirical study, *i.e.,* Apache Cassandra 1.1. When considering the God Class smell, unlike our tool, none of the available tools is able to identify all the eight actual smell instances we found by manually analyzing the classes of this system. Indeed, DECOR identifies only one of the actual instances, while JDeodorant and HIST detect three of them. Therefore, the use of existing tools would have resulted in a less comprehensive analysis. Of course, our tool pays the higher recall with a lower precision with respect to other tools. However, this is not a threat for our study, because the manual validation conducted on the instances automatically

detected by the tool aims at discarding the false positives, while keeping the true positive smell instances.

Once collected the data about the presence of each of the 13 code smells in each of the 395 analyzed software releases we used them to answer our research questions. Concerning **RQ1** we verified what is the diffuseness of the considered code smells in the analyzed systems. We also verified if there is any correlation between systems' characteristics (# Classes, #Methods, and KLOCs) and the presence of code smells in them. To compute the correlation on each analyzed system release we used the Spearman rank correlation analysis [159] between the different systems' characteristics and the presence of code smells in them. Such an analysis measures the strength and direction of association between two ranked variables, and ranges between -1 and 1, where 1 represents a perfect positive linear relationship, -1 represents a perfect negative linear relationship, and values in between indicate the degree of linear dependence between the considered distributions. Cohen [159] provided a set of guidelines for the interpretation of the correlation coefficient. It is assumed that there is no correlation when $0 \leq \rho < 0.1$, small correlation when $0.1 \leq \rho < 0.3$, medium correlation when $0.3 \leq \rho < 0.5$, and strong correlation when $0.5 \leq \rho \leq 1$. Similar intervals also apply for negative correlations.

To answer **RQ2** we mined the change history of the 30 systems object of our study. In particular, to compute the class change-proneness, we extracted the change logs from their versioning systems in order to identify the set of classes modified in each commit. Then, we computed the change-proneness of a class C_i in a release r_j as:

$$change\_proneness(C_i, r_j) = \#Changes(C_i) \tag{4.1}$$

where $\#Changes(C_i)$ is the number of changes performed to C_i by developers during the evolution of the system between the r_{j-1}'s and the r_j's release dates. Note that in the context of this chapter, the changes are the number of commits in which a certain class has been involved, while the releases are all the major releases reported in the repositories of the considered systems.

As for the fault-proneness, we developed a mining tool to extract the bugs fixed for each class of the object systems during their change history. All considered systems exploit Bugzilla[2] or Jira[3] as issue tracker. Firstly, we identified bug fixing commits by mining regular expressions containing issue IDs in the change log of the versioning system, *e.g., "fixed issue #ID"* or *"issue ID"*. Secondly, for each issue ID related to a commit, we downloaded the corresponding issue reports from their issue tracking system and extracted the following information from them: (i) *product name*; (ii) *issue type*, *i.e.,* whether an issue is a bug, enhancement request, etc; (iii) *issue status*, *i.e.,* whether an issue was closed or not; (iv) *issue resolution*, *i.e.,* whether an issue was resolved by fixing it, or whether it was a duplicate bug report, or a "works for me" case; (v) *issue opening date*; (vi) *issue closing date*, if available.

Then, we checked each issue report to be correctly downloaded (*e.g.,* the issue ID identified from the versioning system commit note could be a false positive). After that, we used the issue type field to classify the issue and distinguish bug fixes from other issue types (*e.g.,* enhancements). Finally, we only considered bugs having *Closed* status and *Fixed* resolution. In this way, we restricted our attention to (i) issues that were related to bugs, and (ii) issues that were neither duplicate reports nor false alarms. Having bugs linked to the commits fixing them allowed us to identify which classes were modified to fix each bug. Thus, we computed the fault-proneness of a class C_i in a release r_j as the number of bug fixing activities to which it has been subjected in the period of time between the r_{j-1} and the r_j release dates.

Once extracted all the required information, we compare the distribution of the change- and fault-proneness of classes affected and not by code smells. In particular, we present boxplots of the change- and fault- proneness distributions of the two sets of classes and we also statistically compare them through the Mann-Whitney test [142], a non-parametric test used to evaluate the null hypothesis stating that it is equally likely that a randomly selected value from one sample

[2] `http://www.bugzilla.org`

[3] `https://www.atlassian.com/software/jira`

will be less than or greater than a randomly selected value from a second sample. The results are intended as statistically significant at $\alpha = 0.05$. We estimated the magnitude of the measured differences by using the Cliff's Delta (or d), a non-parametric effect size measure [143] for ordinal data. We followed well-established guidelines to interpret the effect size values: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [143].

It is important to note that the analysis of the fault-proneness might be biased by the fact that a bug might have been introduced before the introduction of the code smell. This would lead to an overestimation of the actual number of bug fixing activities performed on smelly classes in the time period between the releases r_{j-1} and r_j. For this reason, we also analyze the fault-proneness of smelly classes when only considering bug fixing activities related to bugs introduced after the smell introduction. More formally, we computed the fault-proneness of a smelly class C_i in a release r_j as the number of changes to C_i aimed at fixing a bug introduced after the code smell introduction in the period between r_{j-1} and r_j.

To estimate the date in which a bug was likely introduced[4], we exploited the SZZ algorithm[5] [156], which is based on the annotation/blame feature of versioning systems. In summary, given a bug-fix identified by the bug ID, k, the approach works as follows:

1. For each file f_i, $i = 1 \ldots m_k$ involved in the bug-fix k (m_k is the number of files changed in the bug-fix k), and fixed in its revision *rel-fix*$_{i,k}$, we extract the file revision just *before* the bug fixing (*rel-fix*$_{i,k} - 1$).

2. Starting from the revision *rel-fix*$_{i,k} - 1$, for each source line in f_i changed to fix the bug k the *blame* feature of *Git* is used to identify the file revision where the last change to that line occurred. In doing that, blank lines and lines that only contain comments are identified using an island grammar parser [9]. This produces, for each file f_i, a set of $n_{i,k}$ fix-inducing revisions *rel-bug*$_{i,j,k}$,

[4]The right terminology is "when the bug induced the fix" because of the intrinsic limitations of the SZZ algorithm, which cannot precisely identify whether a change actually introduced the bug.

[5]SZZ stays for the last name initials of the three algorithm's authors.

$j = 1 \dots n_{i,k}$. Thus, more than one commit can be indicated by the SZZ algorithm as responsible for inducing a bug.

By adopting the process described above we are able to approximate the time periods in which each class was affected by one or more bugs. We excluded from our analysis all the bugs occurring in a class C_i before it became smelly.

Note that in the context of **RQ2** we considered all classes of the object systems: if a class was smelly on some releases and not-smelly on other releases, it contributes to both sets of smelly (for the software releases in which it was smelly) and non-smelly (for the software releases in which it was not smelly) classes. Also, in this research question we do not discriminate the specific type of smell affecting a class (*i.e.,* a class is considered smelly if it contains any type of code smell). A fine grained analysis of the impact of the different smell types of the class change- and fault-proneness is presented in the next research question.

In **RQ3** we exploited the code smells' oracle we built in the context of our first study (*i.e.,* the one reporting the code smells affecting each class in each of the 395 considered releases) to identify in which releases of each system a class was smelly or not smelly. Then, we focused our attention only on classes affected by at least one smell instance in at least one of the analyzed software releases but not in all of them. In this way, we can compare their change- and fault-proneness when they were affected and not affected by smells. Note that in this case we consider each smell in isolation. For example, suppose that a class C was firstly affected by the God Class smell between releases r_i and r_{i+1}. Then, the smell was not detected between releases r_{i+1} and r_{i+2}. Finally, the smell re-appeared between releases r_{i+2} and r_{i+3}. We compute the change-proneness of C when it was smelly by summing up the change-proneness of C in the periods between r_i and r_{i+1} and between r_{i+2} and r_{i+3}. Similarly, we computed the change-proneness of C when it was non-smelly by computing the change-proneness of C in the period between r_{i+1} and r_{i+2}. Following the same procedure, we compare the fault-proneness of classes when they were affected and not by a code smell. As done for **RQ2**, the comparison is performed by using boxplots and statistical tests for significance

(Mann-Whitney test) and effect size (Cliff's Delta).

4.3 Analysis of the Results

In this section we answers our three research questions.

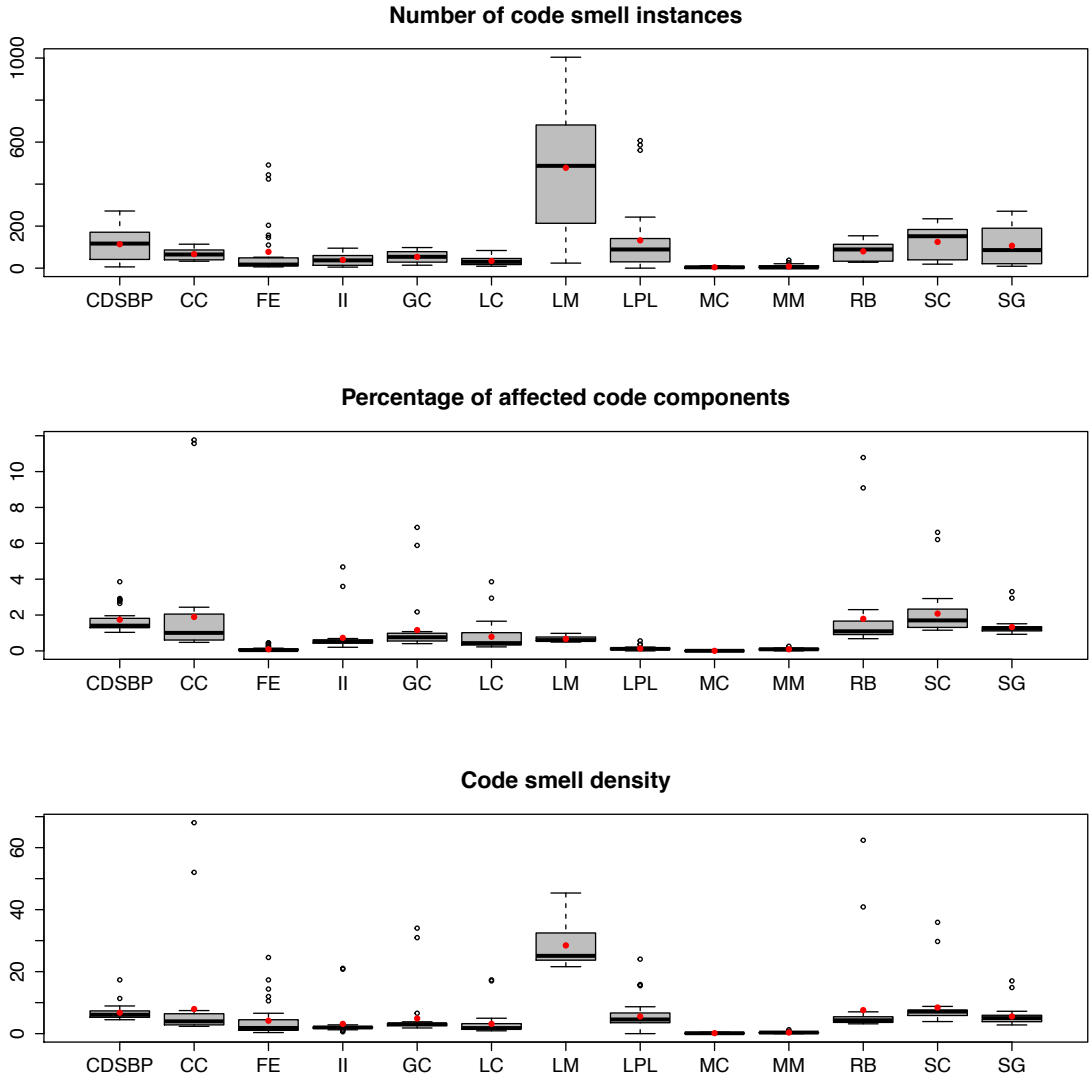


Figure 4.1: Absolute number, percentage, and density of code smell instances in the analyzed systems.

4.3.1 Diffuseness of code smells (RQ1)

Fig. 4.1 shows the boxplot reporting (i) the absolute number of code smell instances, (ii) the percentage of affected code components (*i.e.*, percentage of affected classes/methods[6]), and (iii) the code smell density (*i.e.*, number of code smells per KLOC) affecting the software systems considered in our study. Note that for the sake of clarity, we aggregate the results considering all the systems as a unique dataset.

The boxplots highlight significant differences in the diffusiness of code smells. The first thing that leaps to the eyes is that code smells like *Feature Envy*, *Message Chain*, and *Middle Man* are poorly diffused in the analyzed systems. For instance, across the 395 system releases, the highest number of *Feature Envy* instances in a single release (a Xerces release) is 17, leading to a percentage of affected methods of only 2.3%. We found instances of *Feature Envy* in 50% of the analyzed 395 releases.

The *Message Chain* smell is also poorly diffused. It affects 13% of the analyzed releases and, in the most affected release (a HSQLDB's release), only four out of the 427 classes (0.9%) are instances of this smell. Note that in previous work *Message Chain* resulted to be the smell having the highest correlation with fault-proneness [17]. Therefore, the observed results indicate that, although the *Message Chain* smell is potentially harmful, its diffusion is fairly limited.

Finally, the last poorly diffused code smell is the *Middle Man*. 30% of the 395 analyzed releases are affected by at least one of its instances, with eight being the highest number of instances in a single release (a Cassandra release). In particular, the classes affected by the *Middle Man* in Cassandra 0.6 were 8 out of 261 (3%). In this case, all identified *Middle Man* instances affect classes belonging to the `org.apache.cassandra.utils` package, grouping together classes delegating most of their work to classes in other packages. For example, the `HintedHandOffManager` class delegates eleven out of the twelve methods it contains to the `StorageService` class from the `apache.cassandra.service`

[6]Depending on the code smell granularity, we report the percentage of affected classes or methods.

package.

Other code smells are instead quite diffused. For example, we found at least one instance of *Long Method* in 84% of the analyzed releases (331 out of 395). In particular, each of these 331 releases is affected, on average, by 44 *Long Method* instances with the peak of 212 in an `Apache Derby`'s release[7]. We manually analyzed that release (*i.e.*, `10.1`) to understand the reasons behind the presence of so many *Long Method* instances. Most of the instances are in the `impl.sql.compile` package, grouping together classes implementing methods responsible for the parsing of code statements written by using the SQL language. Such parsing methods are in general very complex and long (on average, 259 LOC). For a similar reason, we found several instances of *Long Method* in `Eclipse Core`. Indeed, it contains a high number of classes implementing methods dealing with the parsing of the code in the IDE. While we cannot draw any clear conclusion based on the manual analysis of these two systems, our feeling is that the inherent complexity of such parsing methods makes it difficult for developers to (i) write the code in a more concise way, avoiding *Long Method* code smells, or (ii) remove the smell by applying, for instance, extract method refactoring.

Another quite diffused code smell is the *Spaghetti Code*, present in 83% of the analyzed releases (327 out of 395) with the highest number of instances (54) found in a `JBoss`'s release. Other diffused code smells are *Speculative Generality* (80% of affected releases), *Class Data Should Be Private* (77%), *Inappropriate Intimacy* (71%), and *God Class* (65%).

Interestingly, the three smallest systems considered in our study (`Hibernate`, `jSL`, and `Sax`) do not present any instance of code smell in any of the 31 analyzed releases. This result might indicate that in small systems software developers are generally able to better keep under control the code quality, avoiding the introduction of code smells. To further investigate this point we computed the correlation between systems' size (in terms of # Classes, #Methods, and LOCs) and the number of instances of each code smell (see Table 4.4). As expected, some code smells have a positive correlation with the size attributes, meaning that the larger the

[7]Apache Derby is an open source relational database.

Table 4.4: Correlation between code smell instances and system size.

Code smell	ρ with #Classes	ρ with #Methods	ρ with LOCs
Class Data Should Be Private	**0.72**	**0.82**	**0.82**
Complex Class	*0.49*	**0.71**	**0.73**
Feature Envy	-0.07	-0.02	0.01
God Class	**0.50**	**0.76**	**0.82**
Inappropriate Intimacy	-0.02	0.02	0.08
Lazy Class	0.20	*0.32*	*0.32*
Long Method	*0.47*	**0.72**	**0.79**
Long Parameter List	-0.12	-0.09	-0.05
Message Chain	-0.10	-0.03	0.03
Middle Man	0.07	0.19	0.18
Refused Bequest	**0.74**	**0.82**	**0.81**
Spaghetti Code	**0.69**	**0.74**	**0.75**
Speculative Generality	**0.85**	**0.78**	**0.77**

In *Italic* the medium correlations, in **bold** the strong correlations

system, the higher the number of code smell instances in it. There are also several code smells for which this correlation does not hold (*i.e., Feature Envy*, *Inappropriate Intimacy*, *Long Parameter List*, *Message Chain*, and *Middle Man*). With the exception of *Long Parameter List*, all these smells are related to "suspicious" interactions between the classes of the system (*e.g.,* the high coupling represented by the *Inappropriate Intimacy* smell). It is reasonable to assume that the interactions of such classes is independent from the systems' size and mainly related to correct/wrong design decisions.

We also compute the code smell density as the number of smell instances per KLOC in each of the 395 analyzed releases (see bottom part of Fig. 4.1). The results confirm that the *Long Method* is the most diffused smell, having the the highest average density (*i.e.,* 28 instances per KLOC). Also *Refused Bequest* and *Complex Class* smells, *i.e.,* the code smells having the highest percentage of affected

Table 4.5: **RQ1**: Diffuseness of the studied code smells.

Code smell	% affected releases	avg. number of instances	max number of instances	Diffuseness
Long Method	84%	44	212	High
Spaghetti Code	83%	12	54	High
Speculative Generality	80%	11	65	High
Class Data Should Be Private	76%	12	65	High
Inappropriate Intimacy	71%	4	34	High
God Class	65%	5	26	Medium
Refused Bequest	58%	11	55	Medium
Complex Class	56%	9	35	Medium
Long Parameter List	47%	16	77	Medium
Feature Envy	50%	3	17	Low
Lazy Class	47%	5	21	Low
Middle Man	30%	2	8	Low
Message Chain	13%	2	4	Low

code components, are confirmed to be quite diffused in the studied systems. All the other smells seem to have diffuseness trends similar to the ones previously discussed.

Table 4.5 classifies the studied code smells on the basis of their diffuseness in the releases subject of our study. The "% of affected releases" column reports the percentage of analyzed releases in which we found at least one instance of a specific smell type. For example, a smell like *Long Method* affects 84% of releases, *i.e.*, 395*0.84=332 releases.

4.3.2 Change- and fault-proneness of classes affected/not affected by code smells (RQ2)

Fig. 4.2 shows the boxplots of change-proneness for classes affected/not affected by code smells. Our results confirm the findings reported by Khomh *et al.* [17], showing that classes affected by code smells have a higher change-proneness than

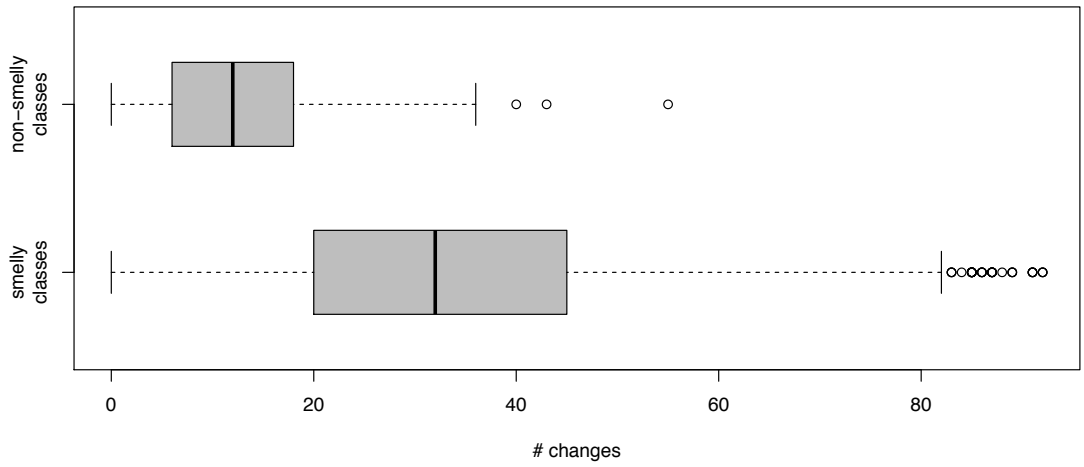


Figure 4.2: Change-proneness of classes affected and not by code smells

other classes. Indeed, the median change proneness for classes affected by code smells (32) is almost three times higher with respect to the median change proneness of the other classes (12). For example, the `Eclipse` class `IndexAllProject` affected by the *Long Method* smell (in its `execute` method) has been modified 77 times during the time period between the release 8 (2.1.3) and 9 (3.0), while the median value of changes for classes not affected by any code smell is 12. Looking closer to the `execute` method, we found that during the change history of the system its number of lines varied between 671 and 968 due to the addition of several features. The results of the Mann-Whitney and Cliff tests highlight a statistically significant difference in the change-proneness of classes affected and not affected by code smell (p-value$<$0.001) with a large effect size (d=0.68).

Also in the case of the fault-proneness the results show important differences between classes affected and not affected by code smells, even if such differences are less marked than those observed for the change-proneness (see Fig. 4.3). The median value of the number of bugs fixed on classes not affected by smells is 3 (third quartile=5), while the median for classes affected by code smells is 9—+300%—(third quartile=12). The results confirm what already observed by Khomh

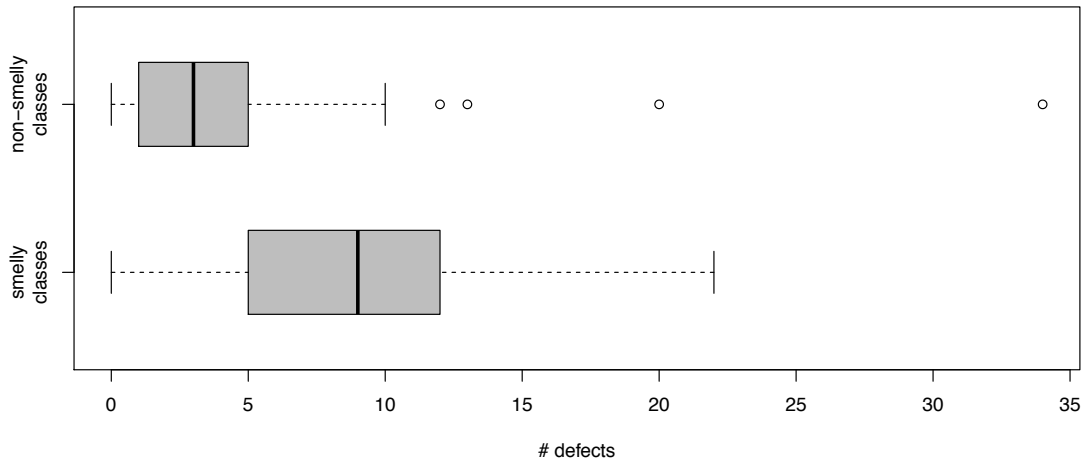


Figure 4.3: Fault-proneness of classes affected and not affected by code smells.

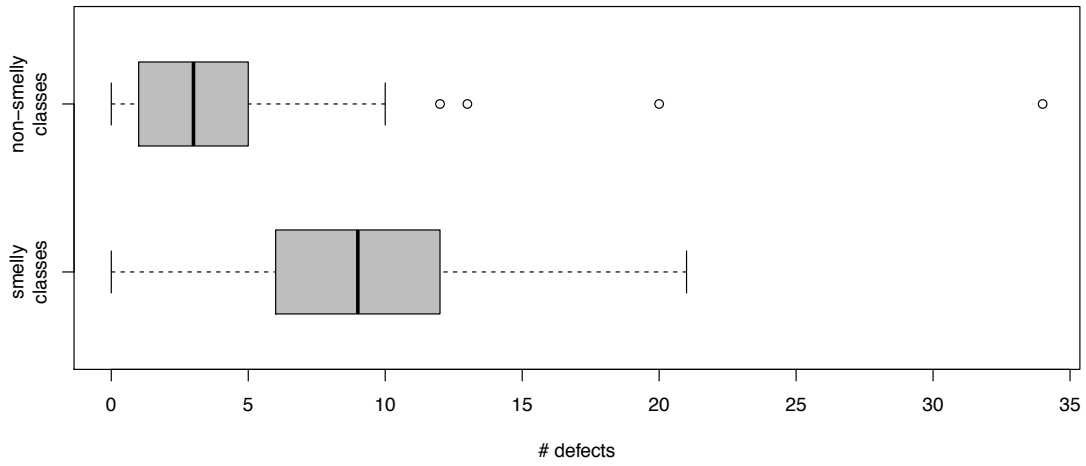


Figure 4.4: Fault-proneness of classes affected and not affected by code smells when considering the bugs introduced after the smell introduction only.

et al. [17]. The observed difference is statistically significant (p-value$<$0.001) with a medium effect size (d=0.41).

When only considering the bugs induced after the smell introduction, the re-

sults still confirm previous findings. Indeed, as shown in Fig. 4.4, smelly classes still have a much higher fault-proneness with respect to non-smelly classes. In particular, the median value of the number of bugs fixed in non-smelly classes is 2 (third quartile=5), as compared to the 9 of smelly classes (third quartile=12). The difference is statistically significant (p-value$<$0.001) with a large effect size (d=0.82).

This result can be explained by the findings reported in the work by Tufano *et al.* [47], in which the authors show that most of the smells are introduced during the very first commit involving the affected class (*i.e.*, when the class is added for the first time to the repository). As a natural implication, most of the bugs introduced in smelly classes are introduced after the code smell appearance (because in most of cases, the smell is there since the class creation). This conclusion is also supported by the fact that in our dataset only 21% of the bugs related to smelly classes are introduced before the smell introduction.

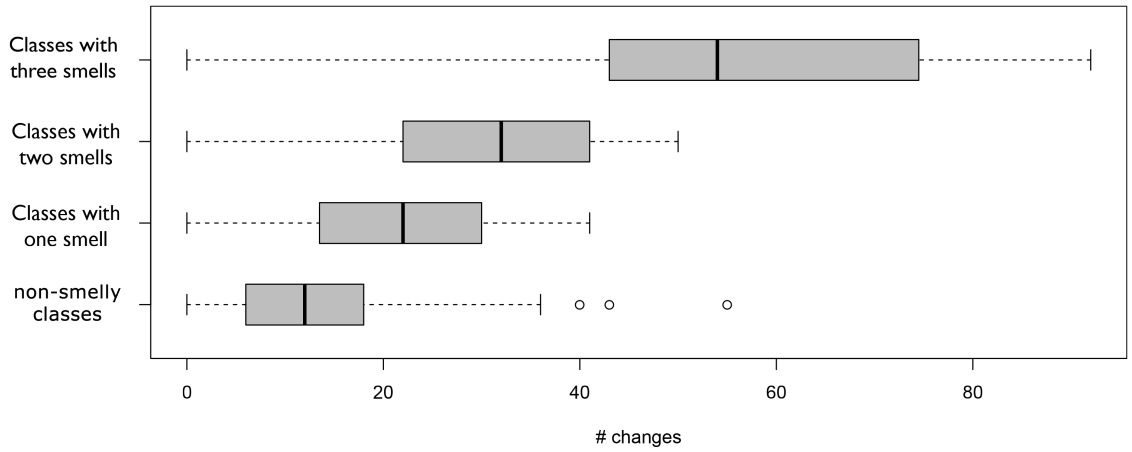


Figure 4.5: Change-proneness of classes affected by different number of code smells.

While the analysis carried out until now clearly highlighted a trend in terms of change- and fault- proneness of smelly and non-smelly classes, it is important to note that a smelly class could be affected by **one or more** smells. For this reason, we performed an additional analysis to verify how the change- and fault-

proneness of classes varies when considering classes affected by zero, one, two, and three code smells. Note that in our dataset there are no classes affected by more than three smells in the same system release. Moreover, if a class was affected by two code smells in release r_{j-1} and by three code smells in release r_j, its change- (fault-) proneness between releases r_{j-1} and r_j contributed to the distribution representing the change- (fault-) proneness of classes affected by two smells while its change- (fault-) proneness between releases r_j and r_{j+1} contributed to the distribution representing the change- (fault-) proneness of classes affected by three smells. Fig. 4.5 reports the change-proneness of the four considered sets of classes, while Fig. 4.6 and Fig. 4.7 depict the results achieved for fault-proneness.

In terms of change-proneness, the trend depicted in Fig. 4.5 shows that the higher the number of smells affecting a class, the higher its change-proneness. In particular, the median number of changes goes from 12 for non-smelly classes, to 22 for classes affected by one smell (+83%), 32 for classes affected by two smells (+167%), and up to 54 for classes affected by three smells (+350%). Table 4.6 reports the results of the Mann-Whitney test and of the Cliff's delta obtained when comparing the change-proneness of these four categories of classes. Since we performed multiple tests, we adjusted our p-values using the Holm's correction procedure [160]. This procedure sorts the p-values resulting from n tests in ascending order, multiplying the smallest by n, the next by $n-1$, and so on.

Table 4.6: Change-proneness of classes affected by a different number of code smells: Mann-Whitney test (adj. p-value) and Cliff's Delta (d).

Test	adj. p-value	d
zero smells *vs* one smell	<0.001	0.53 (Large)
zero smells *vs* two smells	<0.001	0.80 (Large)
zero smells *vs* three smells	<0.001	0.89 (Large)
one smell *vs* two smells	<0.001	0.42 (Medium)
one smell *vs* three smells	<0.001	0.84 (Large)
two smells *vs* three smells	<0.001	0.72 (Large)

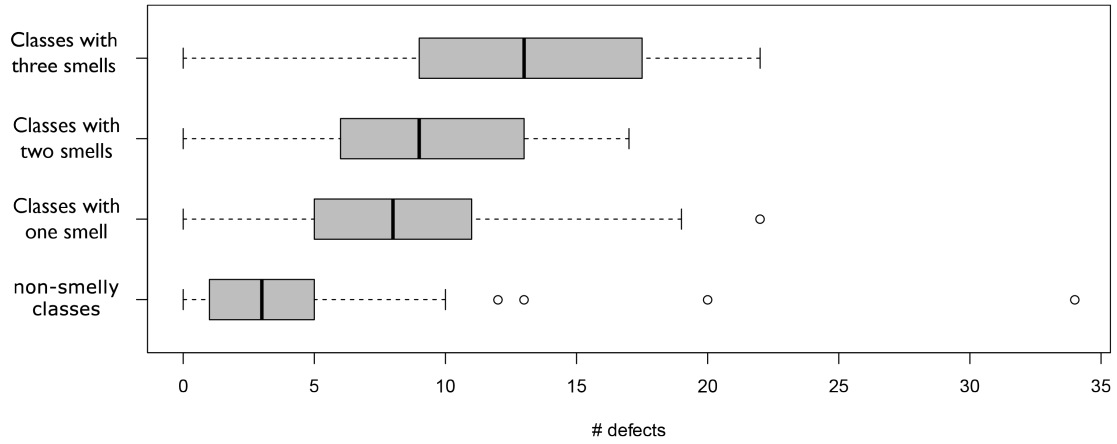


Figure 4.6: Fault-proneness of classes affected by different number of code smells.

The achieved results show that (i) classes affected by a lower number of code smells always exhibit a statistically significant lower change-proneness than classes affected by a higher number of code smells, and (ii) the effect size is always large with the only exception of the comparison between classes affected by one smell and classes affected by two smells, for which the effect size is medium.

Table 4.7: Fault-proneness of classes affected by a different number of code smells: Mann-Whitney test (adj. p-value) and Cliff's Delta (d).

Test	adj. p-value	d
zero smells *vs* one smell	<0.001	0.74 (Large)
zero smells *vs* two smells	<0.001	0.74 (Large)
zero smells *vs* three smells	<0.001	0.89 (Large)
one smell *vs* two smells	<0.001	0.14 (Small)
one smell *vs* three smells	<0.001	0.53 (Large)
two smells *vs* three smells	<0.001	0.40 (Medium)

Similar observations can be made for what concerns the fault-proneness. Fig. 4.6 depicts the boxplots reporting the fault-proneness of classes affected by zero, one, two, and three code smells. With the increase in the number of code smells, the median fault-proneness of the classes grows from 3 for the non-smelly classes

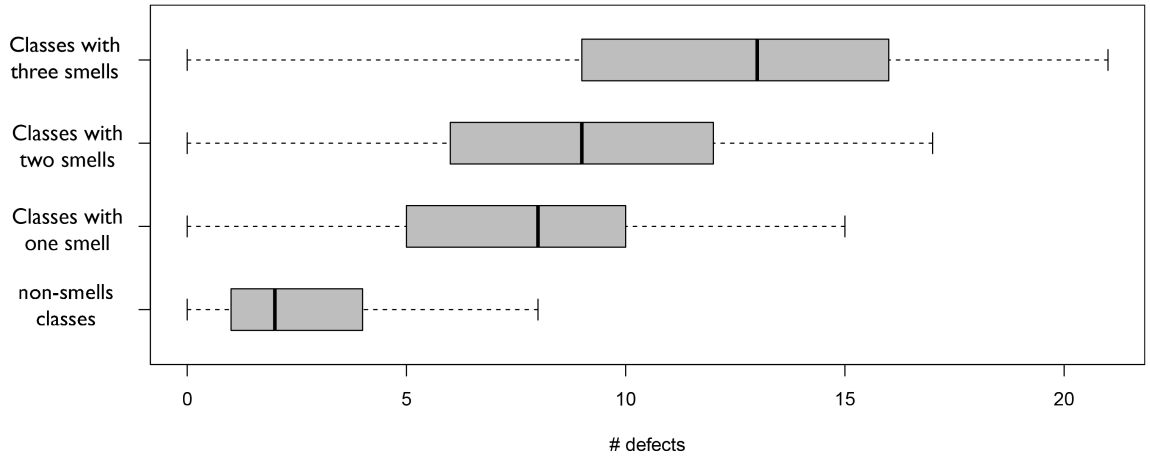


Figure 4.7: Fault-proneness of classes affected by different number of code smells when considering only the bugs induced after the smell introduction.

up to 12 (+300%) for the classes affected by three code smells.

Table 4.8: Fault-proneness of classes affected by a different number of code smells when considering only bugs induced after the smell introduction: Mann-Whitney test (adj. p-value) and Cliff's Delta (d).

Test	adj. p-value	d
zero smells *vs* one smell	<0.001	0.75 (Large)
zero smells *vs* two smells	<0.001	0.71 (Large)
zero smells *vs* three smells	<0.001	0.95 (Large)
one smell *vs* two smells	<0.001	0.19 (Small)
one smell *vs* three smells	<0.001	0.61 (Large)
two smells *vs* three smells	<0.001	0.43 (Medium)

The results of the statistical analysis reported in Table 4.8 confirm the significant difference in the fault-proneness of classes affected by a different number of code smells, with a large effect size in most of the comparisons.

When looking at the boxplots of Fig. 4.7, which refers to the analysis of the fault-proneness performed considering only the bugs introduced after the smell

introduction, we can confirm previous findings. Indeed, the higher the number of code smells affecting a class the higher its fault-proneness. The significant differences are also confirmed by the statistical tests reported in Table 4.8.

4.3.3 Change- and fault-proneness of classes when code smells are introduced and removed (RQ3)

Fig. 4.8 shows, for each considered code smell types, a pair of boxplots reporting the change-proneness of the same set of classes during the time period in which they were affected (*S* in Fig. 4.8) and not affected (*NS* in Fig. 4.8) by that specific code smell.

In all pairs of boxplots a recurring pattern can be observed: when the classes are affected by the code smell they generally have a higher change-proneness than when they are not affected. This result holds for all code smells but *Middle Man* (MM), *Lazy Class* (LC), *Feature Envy* (FE), and *Class Data Should Be Private* (CDSBP).

For classes being affected by a *God Class* (GC) smell we can observe an increase of +283% of the change-proneness median value (46 *vs* 12). The case of the `Base64` class belonging to the *Elastic Search* system is particularly representative: when affected by the *God Class* smell, the developers modified it 87 times on average (the average is computed across the 5 releases in which this class was smelly); instead, when the class was not affected by the code smell, the developers modified it only 10 times on average (the class was not smelly in 3 releases).

Similar results can be observed for the *Complex Class* (CC) smell, with the median change-proneness of classes equal to 55 in the time period in which they are affected by this smell, compared to 34 when they are non-smelly classes. For example, when the `Scanner` class of the *Eclipse Core* project was affected by this smell, it was modified 95 times on average (across the 18 releases in which the class was smelly) as opposed to the 27 average changes observed across the 11 releases in which it was not smelly.

The discussion is quite similar for code smells related to errors in the applications of Object Oriented principles. For example, for classes affected by *Refused*

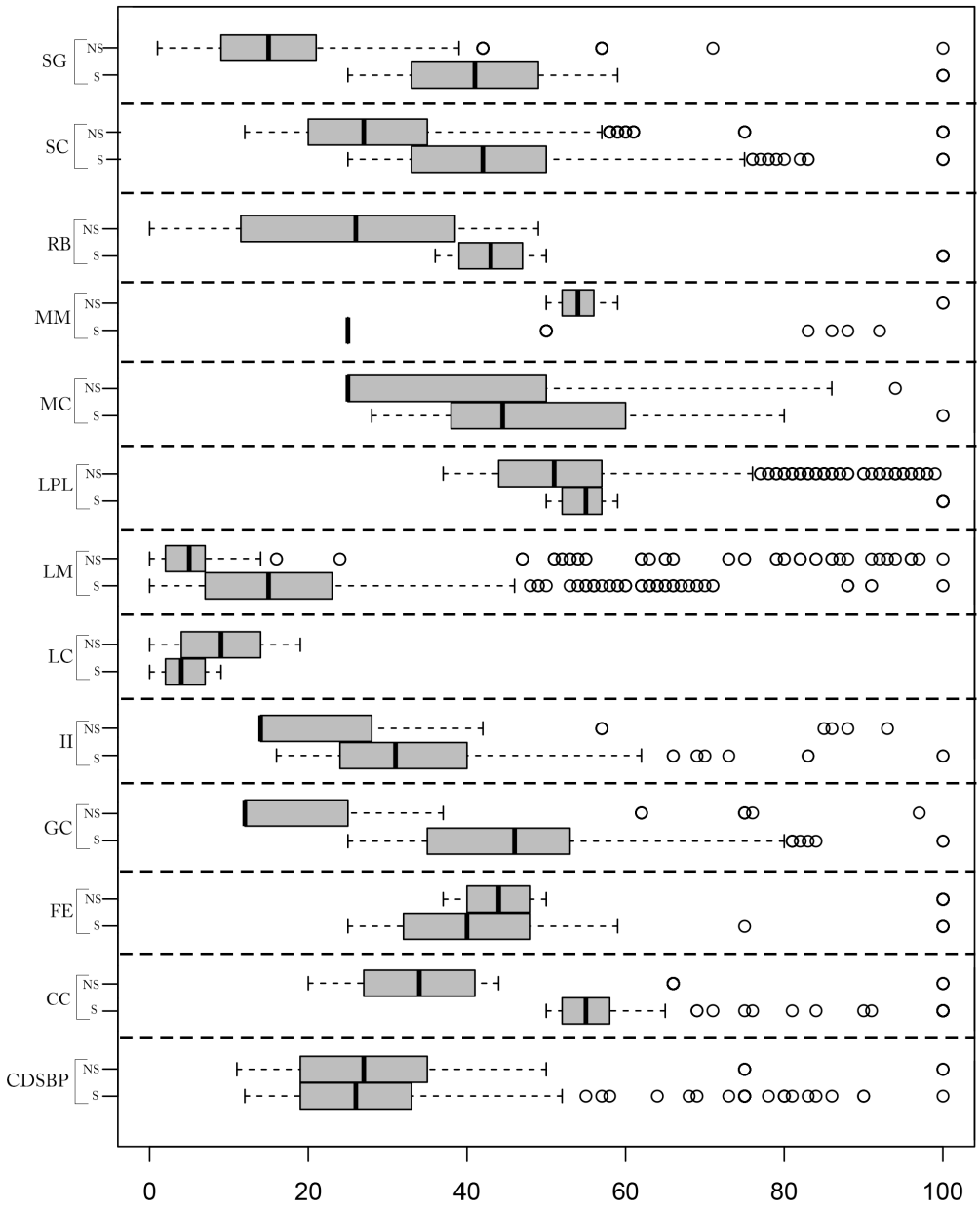


Figure 4.8: Change-proneness of classes affected by a code smell compared to the change-proneness of the same classes during the time period in which they were not affected by a code smell.

Bequest (RB) the median change-proneness goes from 43 (when they are affected) down to 26 (when thet are not affected). The case of the class `ScriptWriterBase`

of the *HSQLDB* project is particularly interesting. This class has been involved in 52 changes, on average, during the time period in which it was affected by RB (13 releases), while the average number of changes decreased to 9 during the time period in which it was not smelly (4 releases).

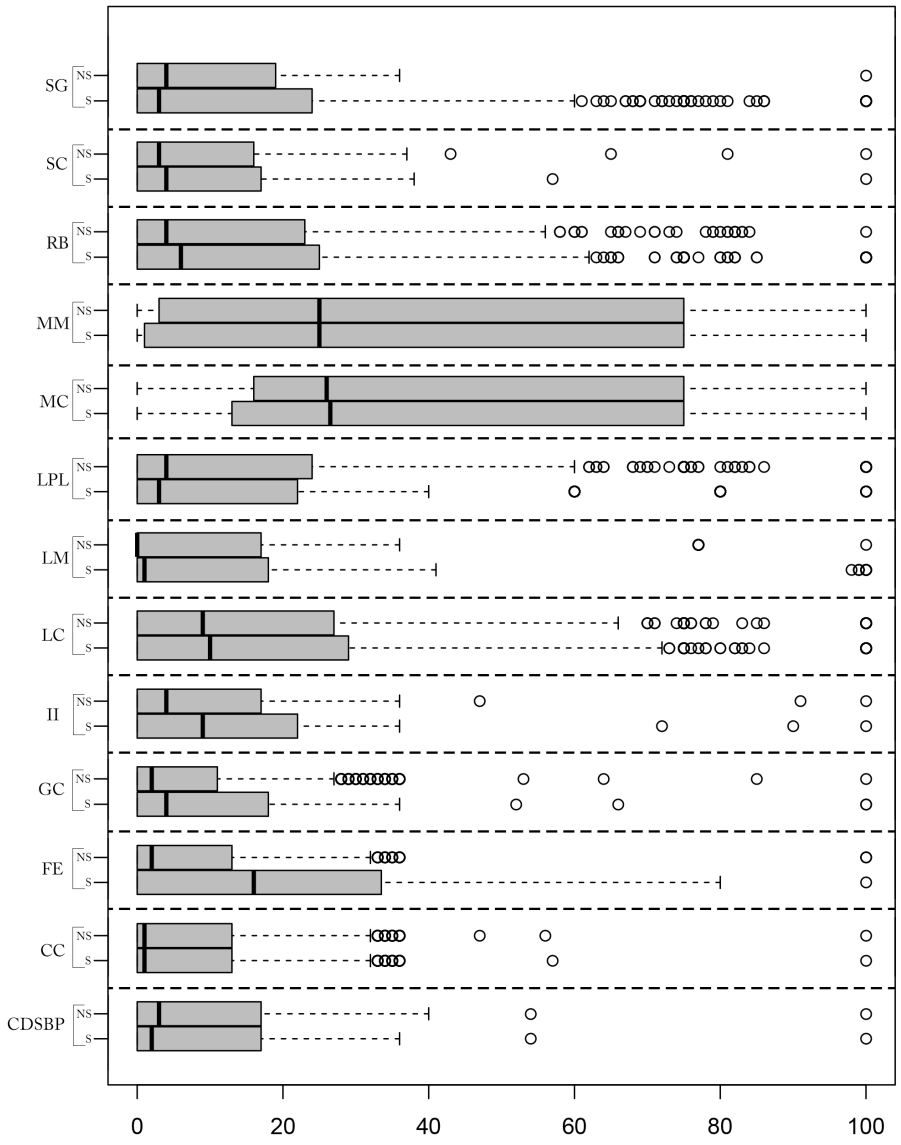


Figure 4.9: Fault-proneness of classes affected by a code smell compared to the fault-proneness of the same classes during the time period in which they were not affected by a code smell.

It is also interesting to understand why some code smells reduce the change-proneness. For the *Lazy Class* smell this result is quite expected. Indeed, by definition, this smell arises when a class has small size, few methods, low complexity and it is used rarely from the other classes; in other words, as stated by Fowler *"the class isn't doing enough to pay for itself"* [8]. Removing this smell could mean increasing the usefulness of the class by implementing, for example, new features in it. This is likely going to increase the class change-proneness. Also, the removal of a *Middle Man* (a class delegating most of its responsibilities) is expected to increase the change-proneness of classes, since the non-smelly class will implement, without delegation, a set of responsibilities that are likely to be maintained by developers, thus triggering new changes.

Results of the fault-proneness are shown in Fig. 4.9. Here, the differences between the time periods the classes are affected and not by code smells are less evident, but still present, especially for *Refused Bequest* (RB), *Inappropriate Intimacy* (II), *God Class* (GC), and *Feature Envy* (FE). The most interesting case is the FE, for which we observed that the fault-proneness increases by a factor of 8 when this code smell affects the classes. A representative example is that of the method `internalGetRowKeyAtOrBefore` of the class `Memcache` of the project *Apache HBase*. This method did not present faults when it was not affected by any smell (*i.e.,* the method was not affected by smell in 4 releases of the system). However, when the method started to be too coupled with the class `HStoreKey`, it was affected by up to 7 faults. The reason for this growth is due to the increasing coupling of the method with the class `HStoreKey`. Indeed, a HBase developer on the issue tracker[8] commented on the evolution of this method: "*Here's a go at it. The logic is much more complicated, though it shouldn't be too impossible to follow*".

For all other smells we did not observe any strong difference in the fault-proneness of the classes when comparing the time periods during which they were affected and not affected by code smells. While this result might seem a contradiction with respect to what observed in RQ2 and in the previous study by Khomh *et al.* [17], our interpretation is that classes that have been fault-prone in the past will

[8]https://issues.apache.org/jira/browse/HBASE-514

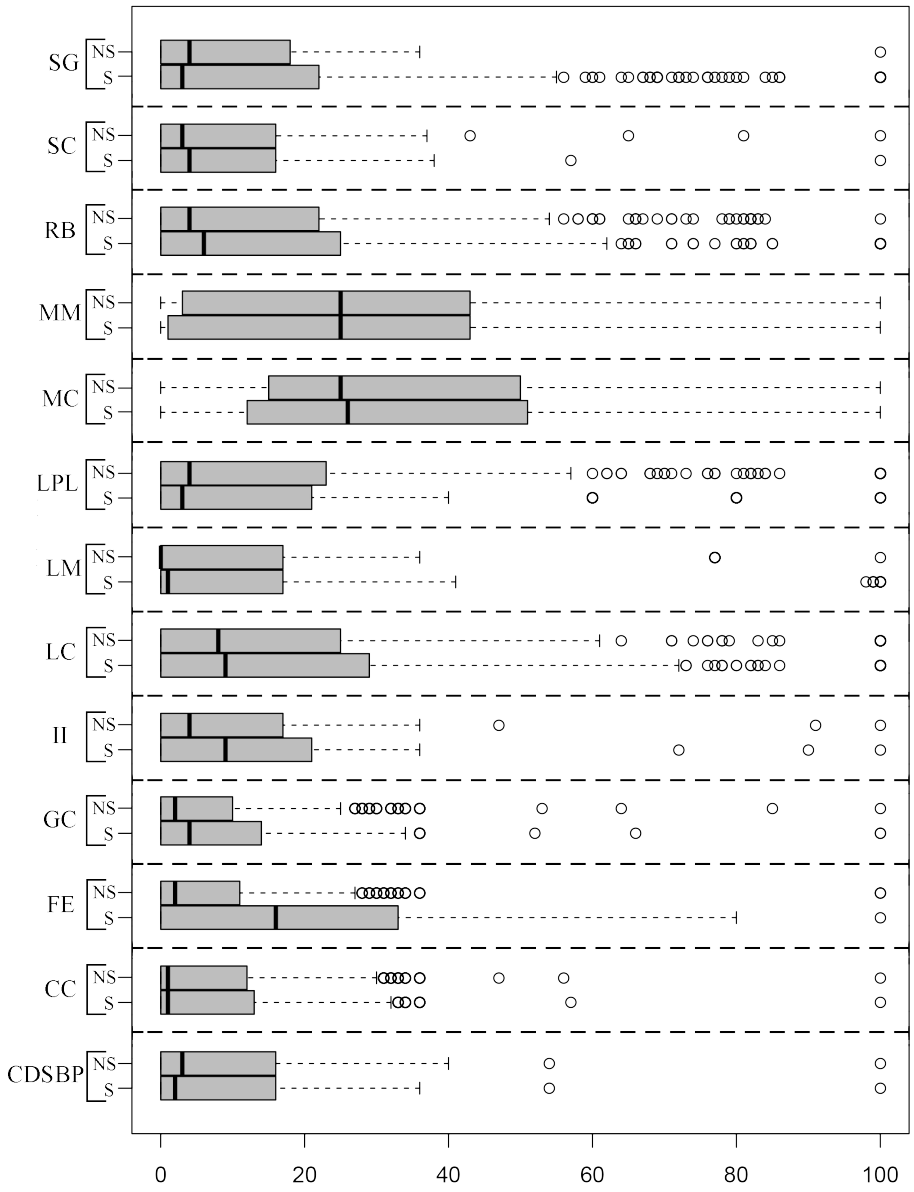


Figure 4.10: SZZ Analysis: Fault-proneness of classes affected by a code smell compared to the fault-proneness of the same classes during the time period in which they were not affected by a code smell.

still continue to be fault-prone, even if a smell has been removed. Moreover, since a smell removal requires a change to the code, it can have side effects like any other change, thus possibly affecting the fault-proneness independently on the smell.

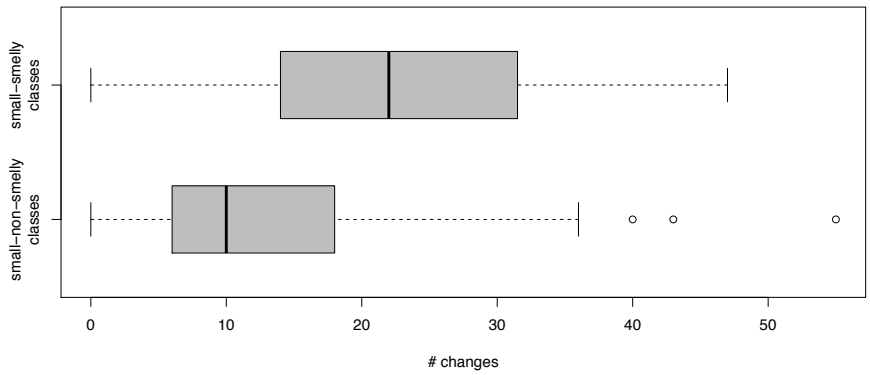
Table 4.9: ORs of independent factors when building logistic model. Statistically significant ORs are reported in bold face.

Dependent Variable	Smell Presence	Size	Their Interaction
Change-proneness	**4.46**	**1.7**	**8.41**
Defect-proneness	1.74	0.93	**2.11**

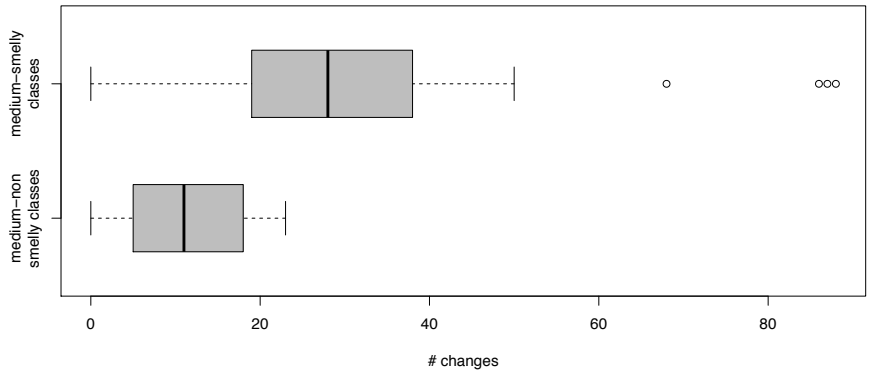
This is also in agreement with previous studies that used the past fault-proneness history of classes to predict their future faults [161]. In essence, there seems to be no direct cause-effect relationships between the presence of code smells and the class fault-proneness. Instead, those are two different negative phenomena that tend to occur in some classes of a software project.

When analyzing only the bugs introduced after the smell appearance (Fig. 4.10), we can observe that also in this case the results are in line with those reported above. Indeed, there are no relevant changes between the findings achieved using or not such a filtering (based on the SZZ algorithm). As explained before, this is simply due to the fact that most of the code smells are introduced during the first commit of a class in the repository.

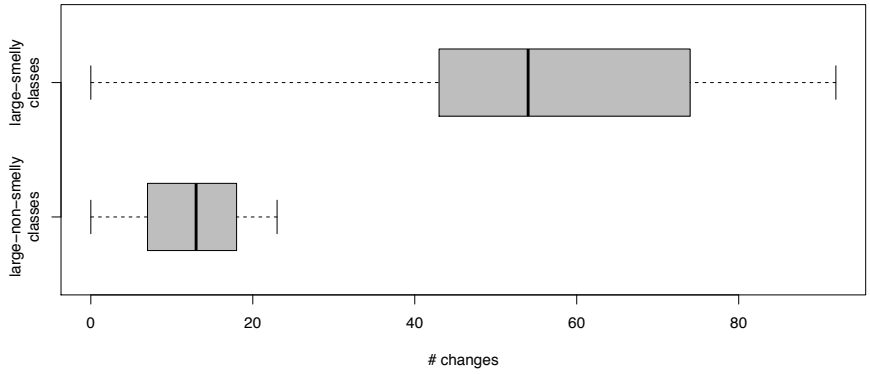
Finally, it is important to point out that our analyses might be influenced by several confounding factors. For instance, it is likely that larger classes are more likely to change over time and to be subject to bug-fix activities. To verify the influence of the size attribute on the results achieved in the context of **RQ2** and **RQ3** we built logistic regression models [162] relating the two phenomena, *i.e.,* change- and fault-proneness, with independent variables represented by the presence of a smell, the size of the component, and their interaction. Table 4.9 reports the ORs achieved from such an analysis. Statistically significant values, *i.e.,* those for which the p-value is lower than 0.05, are reported in bold face. From this analysis, we can notice that the presence of code smells is significantly related to the increase of change-proneness. The size of code components also affects this phenomenon, even if it is to a lower extent, while the interaction of smell presence and size has a strong impact on the change-proneness. In terms of fault-proneness, only the



(a) Change-proneness of smelly and small classes

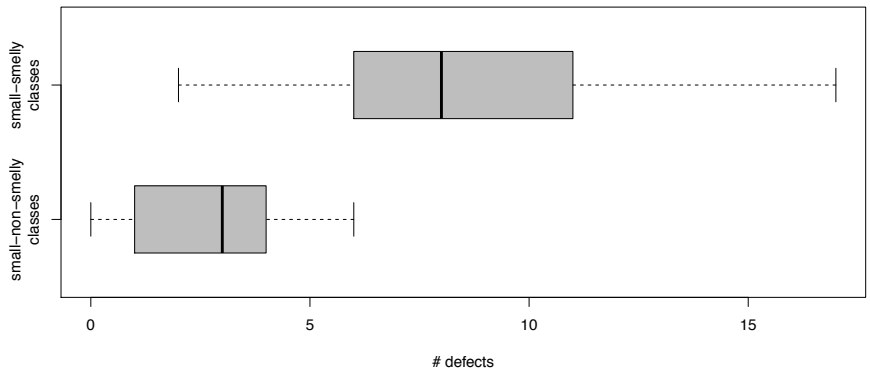


(b) Change-proneness of smelly and medium classes

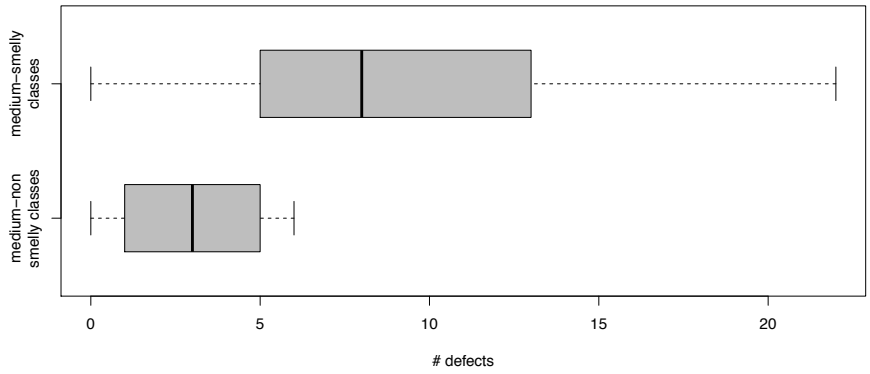


(c) Change-proneness of smelly and large classes

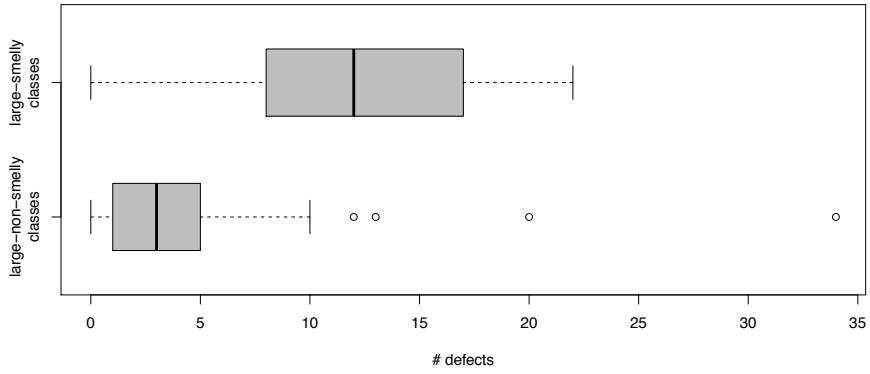
Figure 4.11: Change-proneness of smelly classes grouped by their size.



(a) Fault-proneness of smelly and small classes



(b) Fault-proneness of smelly and medium classes



(c) Fault-proneness of smelly and large classes

Figure 4.12: Fault-proneness of smelly classes grouped by their size.

interaction between the independent variables is statistically significant. This confirms what we observed in RQ3: Code smells are not necessarily the direct cause of the class fault-proneness.

Moreover, to be sure that the results achieved in the context of **RQ2** and **RQ3** were not simply due to a reflection of code size, we re-ran our analysis by considering the change- and the fault-proneness of smelly and non-smelly classes having different size. In particular:

1. We grouped together *smelly* classes with similar size by considering their distribution in terms of size. Specifically, we compute the distribution of the lines of code of classes affected by code smells. This first step results in the construction of (i) the group composed by all the classes having a size lower than the first quartile of the distribution of the size of the classes, *i.e., small* size; (ii) the group composed by all the smelly classes having a size between the first and the third quartile of the distribution, *i.e., medium* size; and (iii) the group composed by the smelly classes having a size larger than the third quartile of the distribution of the size of the classes, *i.e., large* size;

2. We applied the same strategy for grouping *small*, *medium*, and *large* non-smelly classes; and

3. We computed the change- and the fault-proneness for each class belonging to the six groups, in order to investigate whether smelly-classes are more change- and fault-prone regardless from their size.

The obtained results are shown in Figures 4.11 and 4.12. As it is possible to observe, the presence of code smells increase both the change- and fault-proneness of classes, regardless from their size. Thus, we can confirm the findings shown before and claim that code smells actually represent a serious threat for the maintainability of software systems.

4.4 Threats to Validity

This section discusses the threats that could affect the validity of our study.

The main threat related to the relationship between theory and observation (*construct validity*) are due to imprecisions/errors in the measurements we performed. Above all, we relied on a tool we built and made publicly available in our online appendix [157] to detect candidate code smell instances. Our tool exploits conservative detection rules aimed at ensuring high recall at the expense of low precision. Then, two of the authors manually validated the identified code smells to discard false positives. Still, we cannot exclude the presence of false positives/negatives in our dataset.

We assessed the change- and fault-proneness of a class C_i in a release r_j as the number of changes and the number of bug fixes C_i was subject to in the time period t between the r_j and the r_{j+1} release dates. This implies that the t's length could play a role in the change- and fault-proneness of classes (*i.e.*, the longer t, the higher the classes' change- and fault-proneness). However, it is worth noting that:

1. This holds for both smelly and non-smelly classes, thus reducing the bias of t as a confounding factor.

2. To mitigate such a threat we completely re-run our analysis by considering a normalized version of class change- and fault-proneness. In particular, we computed the change-proneness of a class C_i in a release r_j as:

$$change\_proneness(C_i, r_j) = \frac{\#Changes(C_i)}{\#Changes(r_j)}$$

where $\#Changes(C_i)$ is the number of changes performed to C_i by developers during the evolution of the system between the r_{j-1}'s and the r_j's release dates and $\#Changes(r_j)$ is the total number of changes performed on the whole system during the same time period. In a similar way, we computed the fault-proneness of a class C_i in a release r_j as:

$$fault\_proneness(C_i, r_j) = \frac{NOBF(C_i)}{NOBF(r_j)}$$

where $NOBF(C_i)$ is the number of bug fixing activities performed on C_i by developers between the r_{j-1}'s and the r_j's release dates and $NOBF(r_j)$ is the total number of bugs fixed in the whole system during the same time period.

The achieved results are consistent with what observed in Section 4.3.

In addition, we cannot exclude imprecisions in the classes' fault-proneness measurement due to misclassification of issues (*e.g.*, an enhancement classified as a bug) in the issue-tracking systems [153]. At least, the systems we consider use an explicit classification of bugs, distinguishing them from other issues.

To investigate whether there is a temporal relationship between the occurrence of a code smell and a bug induction, we relied on the SZZ algorithm [156]. We are aware that such an algorithm only gives a rough approximation of the set of commits inducing a fix, because (i) of the intrinsic limitation of the line-based differencing of git, and (ii) because in some cases a bug can be fixed without modifying the lines inducing it, *e.g.*, by adding a workaround or in general changing the control-flow elsewhere.

The main threats related to the relationship between the treatment and the outcome (*conclusion validity*) might be represented by the analysis method exploited in our study. We discuss our results by presenting descriptive statistics and using proper non-parametric correlation tests (p-values were properly adjusted when multiple comparisons were performed). In addition, the practical relevance of the differences observed in terms of change- and fault-proneness is highlighted by effect size measures.

Threats to *internal validity* concern factors that could influence our observations. The fact that code smells disappear, may or may not be related to refactoring activities occurred between the observed releases. In other words, other changes could have produced such effects. We are aware that we cannot claim a direct cause-effect relation between the presence of code smells and the fault- and change-proneness of classes, which can be influenced by several other factors. In particular, our observations may be influenced by the different development phases encountered over the change history as well as by developer-related factors (*e.g.*, her experience and workload). Also, we acknowledge that such measures could simply reflect the "importance" of classes in the analyzed systems and, in particular, their central role in the software evolution process. For example, we expect classes controlling the business logic of a system to also be the ones

more frequently modified by developers (high change-proneness) and, as a consequence, subject to the introduction of bugs (high fault-proneness). It is possible that such classes are also the ones more frequently affected by code smells, thus implying high change- and fault-proneness of smelly classes. An in-depth analysis of how such factors influencing the change- and fault-proneness of classes is part of our future agenda.

Finally, regarding the generalization of our findings (*external validity*) this is, to the best of our knowledge, the largest study—in terms of number of software releases (395), and considered code smell types (13)—concerning the diffuseness of code smells and their impact on maintainability properties. However, we are aware that we limited our attention only to Java systems, due to limitations of the infrastructure we used (*e.g.,* the code smell detection tool only works on Java code). Further studies aiming at replicating our work on systems written in other programming languages are desirable. Moreover, we focused our attention on open-source systems only, and we cannot speculate about how the results would be different when analyzing industrial systems. Replications of the study in the context of industrial systems may be worthwhile in order to corroborate our findings.

4.5 Conclusion

This chapter reported a large study, conducted on 395 releases of 30 Java open source projects, aimed at understanding the relevance of the code smells in Java open source projects—in terms of smell diffuseness—and the relation between code smells and source code change- and fault-proneness. The study considered 17,350 instances of 13 different types of code smells, firstly detected using a metric-based approach and then manually validated.

The study results, summarized in Table 4.10, highlighted that:

- **Diffuseness of smells.** The most diffused smells are the one related to size and complexity such as *Long Method*, *Spaghetti Code*, and to some extent *Complex Class* or *God Class*. This seems to suggests that a simple metric-based

Table 4.10: Summary of the Achieved Results

Code Smell	Diffuseness	Removal Effect on Change-Proneness	Removal Effect on Fault-Proneness
CDSBP	**High**	Limited	Limited
Complex Class	Medium	**High**	Limited
Feature Envy	Low	Limited	Medium
God Class	Medium	**High**	Limited
Inappropriate Intimacy	**High**	**High**	Medium
Lazy Class	Low	Limited	Limited
Long Method	**High**	**High**	Limited
LPL	Medium	Limited	Limited
Message Chain	Low	Medium	Limited
Middle Man	Low	Limited	Limited
Refused Bequest	Medium	**High**	Limited
Spaghetti Code	**High**	**High**	Limited
Speculative Generality	**High**	**High**	Limited

monitoring of code quality could already give enough indications about the presence of poor design decisions or in general of poor code quality. Smells not related to size like *Message Chains* and *Lazy Class* are less diffused, although there are also cases of such smells with high diffuseness, see for example *Class Data Should Be Private* and *Speculative Generality*.

- **Relation with change- and fault-proneness.** Generally speaking, our results confirm the results of the previous study by Khomh *et al.* [17], *i.e.,* that classes affected by code smells tend to be more change- and fault-prone than others, and that this is even more evident when classes are affected by multiple smells. At the same time, if we analyze the fault-proneness results for specific kinds of smells, we can also notice that high fault-proneness is particularly evident for smells such as *Message Chain* that are not highly diffused.

- **Effect of smell removal on change- and fault-proneness.** Removing code smells is beneficial most of the times for the code change-proneness. On the other side, we found no substantial differences between the fault-proneness of classes in the periods when they were affected by smells and when they

were not (*e.g.,* before the smell introduction, or after the smell removal). This partially contrast the results of the previous studies [17] and seems to indicate that the smell is not the direct cause of fault-proneness but rather a co-occurring phenomenon in some parts of the system that are intrinsically fault-prone for various reasons. This also confirms the principle that a class exhibiting faults in the past is still likely to exhibit faults in the future [161].

Results of our study stress the importance of dealing with code smells and remove them from the system whenever possible. Otherwise, maintainability of the code (at least for what concerns code change- and fault-proneness) could be negatively affected. This clearly implies the availability of accurate smell detection tools and their integration in a development process, *e.g.,* in a continuous integration pipeline.

As for our future research agenda, we will focus on the definition of recommenders able to alert developers about the presence of potential problematic classes based on their (evolution of) change- and fault-proneness and suggest them appropriate refactoring actions. Moreover, we plan to further analyze the factors influencing the change- and fault-proneness of classes.

Chapter 5

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

5.1 Introduction

Bad code smells represent symptoms of poor design and implementation choices [8]. Bad smells are usually introduced in software systems because developers poorly conceived the design of the code component or because they did not care about properly designing the solution due to strict deadlines. *Complex Class*, *i.e.,* a class that contain complex methods and it is very large in terms of LOC; or *God Class*, *i.e.,* a class that does too much/knows too much about other classes, are only some examples of a plethora of bad smells identified and characterized in well-known catalogues [8, 61].

Recent empirical studies showed that code smells hinder comprehensibility [18], and possibly increase change- and fault-proneness [16, 17]. Also, the interaction between different, co-existing code smells can negatively affect maintainability [9]. Hence, there is empirical evidence that code smells have a negative effect on software evolution, and therefore should be carefully monitored and possibly removed through refactoring operations. Thus, a lot of effort has been devoted for

Table 5.1: Bad smells analyzed in our study [8] [61].

Name	Description
Class Data Should Be Private (CDSBP)	A class exposing its attributes
Complex Class (CC)	Classes having high complexity
Feature Envy (FE)	A method making too many calls to methods of another class to obtain data and/or functionality
God Class (GC)	A class having huge dimension and implementing different responsibilities
Inappropriate Intimacy (II)	Two classes exhibiting high coupling between them
Lazy Class (LC)	A very small class that does not do too much in the system
Long Method (LM)	A method having huge size
Long Parameter List (LPL)	A method having a long list of parameters
Middle Man (MM)	A class delegating all its work to other classes
Refused Bequest (RB)	A class inheriting functionalities that it never uses
Spaghetti Code (SC)	A class without structure that declare long methods without parameters
Speculative Generality (SG)	An abstract class that is not actually needed, as it is not specialized by any other class

the definition of approaches aiming at detecting and removing bad code smells [25, 20, 21, 163, 52, 53].

Despite the existing evidence about the negative effects of code smells [18, 17, 9] and the effort devoted to the definition of approaches for detecting and removing them, it is still unclear whether developers would actually consider all bad smells as actual symptoms of wrong design/implementation choices, or whether some of them are simply a manifestation of the intrinsic complexity of the designed solution. In other words, there seem to be a gap between the theory and the practice. For example, a recent study found that some source code files of the Linux Kernel intrinsically have high cyclomatic complexity. However, this is not considered a design or implementation problem by developers [164]. Also, empirical studies indicated that (i) God Classes sporadically changing are not felt as a problem by developers [12]; and (ii) some developers, in particular junior programmers, work better on a version of a system having some classes that centralized the control, *i.e.*, God classes [43]. These results suggest that the presence of bad smells in source code is sometimes tolerable, and part of developers' design choices.

Recently, Yamashita and Moonen [134] performed an exploratory survey aimed at investigating developers knowledge about code smells, by asking questions like "How familiar are you with bad code smells?". Results showed that a large proportion of respondents did not know about bad code smells. While the study of Yamashita and Moonen aimed at investigating to what extent developers had

a theoretical knowledge of code smells (*i.e.,* knowing them from their name and definition), no study so far investigated whether, given a problem instance—that can be brought back to the presence of a bad smell in the code—developers actually perceive the problem as such and whether they associate the problem to the same symptoms explained in the smell definition.

To bridge this gap, we conducted a study aimed at investigating the developers' perception of code smells. First, we identified and manually validated instances of 12 different bad smells in three open source projects.

Then, we provided a questionnaire to the participants where we showed code snippets affected and not affected by bad smells, and asked whether, in the respondents' opinion, the code component has any problem. In case of a positive answer, we asked them to explain what kind of problem they perceived and how severe they judged it. We asked different categories of subjects to participate in the study, namely (i) Master's students, representing a population of subjects pretty knowledgeable about the theoretical concepts of code smells, (ii) industrial developers, *i.e.,* people with experience on real development projects, but not knowing the code being shown; and (iii) developers from the open-source projects in which the bad smells have been collected. In total, we received responses from 34 subjects, and specifically 15 Master's students, 9 industrial developers, and 10 original developers of the studied projects. The data used in our study are publicly available as replication package[1].

5.2 Design of the Empirical Study

The *goal* of the study is to investigate to what extent bad smells reflect developers' perception of poor design and implementation choices and, in this case, what is their perceived severity of the problem. The *quality focus* is source code comprehensibility and maintainability that can be hindered by the presence of bad smells. The *context* of the study consists of: (i) *objects*, *i.e.,* bad smells identified in three software projects; and (ii) *subjects* (hereby referred to as "participants"), *i.e.,*

[1]http://tinyurl.com/o6lk584

Master's students and professional developers providing their opinions about bad smells.

5.2.1 Research Questions

Our study aims at addressing the following two research questions:

- **RQ1:** *To what extent do bad smells reflect developers' perception of design problems?*
- **RQ2:** *What are the bad smells that developers feel as the most harmful?*

In the context of our study, we considered the twelve code smells briefly described in Table 5.1. Our choice of these smells is not random, but guided by the will of considering a mix of bad smells related to complex/large code components (*e.g., Complex Class*, *God Class*) as well as smells related to the non-adoption of good Object-Oriented coding practices (*e.g., Inappropriate Intimacy*, *Refused Bequest*). However, we did not consider smells such as *Divergent Change* or *Parallel Inheritance*, because their full understanding would require a deep knowledge and/or exploration of the system history.

Table 5.2: Characteristics of the object systems.

Project	KLOC	#Classes	#Methods
ArgoUML 0.34	280	1,889	10,450
Eclipse 3.6.1	440	2,181	18,234
jEdit 4.5.1	165	520	5411

5.2.2 Context Selection

The *objects* considered in the study are bad smells identified in three open-source projects, namely `ArgoUML`, `Eclipse`, and `JEdit`. `ArgoUML` is an open-source UML modeling tool while `Eclipse` is a popular Integrated Development Environment supporting different programming languages. Finally, `JEdit` is a text ed-

itor for programmers. Table 5.2 reports the characteristics of the analyzed projects, namely the size in terms of KLOC, number (#) of classes and number of methods.

Table 5.3: Bad Smells Instances Identified in Each System.

Project	CDSBP	CC	FE	GC	II	LC	LM	LPL	MM	RB	SC	SG
ArgoUML	5	4	1	3	4	0	28	0	2	4	15	28
Eclipse	32	35	6	15	7	15	180	0	2	31	24	12
jEdit	7	21	0	6	4	0	33	9	0	3	18	14

To answer our research questions we needed to identify instances of the twelve considered bad smells in the object systems. Unfortunately, since there are no annotated sets of such smells available in literature, we had to manually identify them. A Master's student from the University of Salerno manually identified instances of the considered bad smells in each of the object systems by relying on the definition of the smells reported in the literature. In such a process, the student also relied on metric extractors and on metric-based definitions of code smells, such as the one of DECOR [20]. For example, *God Classes* were identified as large classes implementing several responsibilities and controlling many other objects in the system, while *Long Methods* were simply identified by analyzing the lines of code composing them. The resulting set of smells has been then validated by a second Master's student to verify that all affected code components identified by the first student were correct. Note that, while this does not ensure completeness in the identification of smells, having multiple manual evaluations ensure enough confidence about the absence of false positives, that could instead occur if relying on automatic detection tools. Also, such a multiple evaluation limited the bias in our dataset. For the aim of our study this was exactly what we needed: a set of reliable bad code smells on the object systems. Note that, we did not find instances of all considered smells in each object system. Table 5.3 reports, for each code smell, the number of its instances identified in the object systems.

As summarized in Table 5.4, *participants* involved in the study belong to the following three categories:

Table 5.4: Study participants: invited and actual respondents.

Category	Invited	Answered	Return rate (%)
Original Developers	45	10	22%
Industrial Developers	28	9	32%
Master's students	15	15	100%
Overall	**88**	**34**	**39%**

1. *Developers working on the three open-source systems*. We sent invitations to active developers of the three object systems, identified by analyzing the systems' commit history[2]. In total, we invited 19 developers from ArgoUML, 11 from Eclipse, and 15 from jEdit. We received responses from 4, 3, and 3 developers, respectively. In the following, we will refer to them as *original developers*. Note that each of the original developers was asked to work on tasks related to the code belonging to the system she had worked on only.

2. *Industrial developers*. We invited 28 industrial developers from different application domains having a programming experience ranging between 2 and 15 years and that generally work on Java development. We obtained an answer from 9 of them; each one performed tasks related to all three systems.

3. *Master's students*. We recruited 15 Master's students attending the Advanced Software Engineering course at the University of Salerno (Italy). Students had good knowledge of Object Oriented programming and they attended a seminar of three hours about bad code smells and design problems. All students performed tasks related to all three systems.

The reason for having these different categories of participants is to get the opinion of developers who know the code very well, as well as of outsiders (industrial developers and Master's students) that, while being less knowledgeable about the code, might provide a less biased indication.

[2]We considered developers that performed at least one commit in the last two years.

5.2.3 Study Procedure

The experimental tasks consisted of questionnaires that participants had to answer through a Web application tool named *eSurveyPro*. In these questionnaires we showed to the participants source code snippets (that may or may not contain code smells) and asked questions about whether the code contained possible design/implementation problems, as well as the perceived severity of the problem, if any.

Specifically, given the object system S_i, the following process was performed:

1. For each code smell c_j having at least one instance in S_i, we randomly selected one instance or took the only one available. Note that with "instance" we refer to the code component(s) affected by the smell. For example, it could be a single method affected by the *Long Parameter List* smell as well as a pair of classes affected by the *Inappropriate Intimacy* smell. Note that we did not select code components affected by more than one bad smell, since we want to isolate each smell involved in our study.

2. For each selected smell instance, we created a task composed of the following questions:

   - In your opinion, does this code component[3] exhibit any design and/or implementation problem?
   - If YES, please explain what are, in your opinion, the problems affecting the code component.
   - If YES, please rate the severity of the design and/or implementation problem by assigning a score on the following five-points Likert scale [165]: 1 (very low), 2 (low), 3 (medium), 4 (high), 5 (very high).

3. For each task related to a code component affected by a bad smell, we also instantiated a task—requiring to participants the same answers seen above—

[3]Depending on the code smell object of the question, a code component could be a method, a class, or a pair of classes.

concerning randomly selected code components not affected by any of the code smells considered in our study. This was done to limit the bias in the study, *i.e.*, avoid that participants always indicated that the code contained a problem and the problem was a serious one.

The final questionnaires included 20 tasks for `ArgoUML` (of which 10 related to components affected by bad smells), 22 for `Eclipse` (11 affected by bad smells), and 18 for `JEdit` (9 affected by bad smells). As explained before, the difference in the number of tasks for the three systems is because as shown in Table 5.3, we identified instances of 10 kinds of smells in `ArgoUML`, 11 in `Eclipse` and 9 in `JEdit`.

All participants invited in our study received an e-mail with instructions on how to answer the survey and a link to the website where each participant could log in to visualize and answer the questions. Participants had up to four weeks to complete this survey.

5.2.4 Data Analysis

To answer **RQ1** we compute, for each type of code smell:

1. The percentage of cases the bad smell has been *perceived* by the participants. With *perceived*, we mean cases where participants answered *yes* to the question: "In your opinion, does this code component exhibit any design and/or implementation problem?"

2. The percentage of times the bad smell has been *identified* by the participants. With *identified*, we mean cases where besides perceiving the smell, participants were also able to identify the exact smell affecting the analyzed code components, by describing it when answering to the question "If yes, please explain what are, in your opinion, the problems affecting the code component". Note that we consider a bad smell as *identified* only if the design problems described by the participant are clearly traceable onto the definition of the bad smell affecting the code component. For example, given the follow-

ing bad smell description for the *Feature Envy* bad smell: *"a method making too many calls to methods of another class to obtain data and/or functionality"*, examples of "correct" descriptions of the problem are *"the method is too coupled with the C_i class"*, or *"the method invokes too many methods of the C_i class"* where, C_i is the class *envied* by the method. On the other side, an answer like *"the method performs too many calls"* is not considered enough to mark the bad smell as *identified*.

Performing this analysis for each bad code smell and for each category of participants in our study we should be able to verify (i) what are the most perceived and identified code smells, and (ii) if the participants' experience and system knowledge play a role in the ability of perceiving and identifying code smells.

As for **RQ2**, we exploited the answers to the question "please rate the severity of the coding problem" provided by participants. Answers have been mainly analyzed through descriptive statistics.

5.3 Analysis of the Results

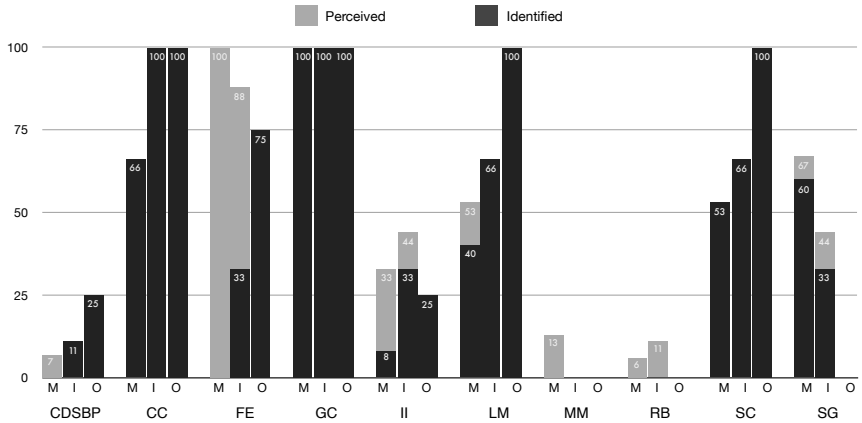


Figure 5.1: Percentage of (M)aster's students, (I)ndustrial developers, and (O)riginal developers that perceived and identified the bad smell examples in ArgoUML.

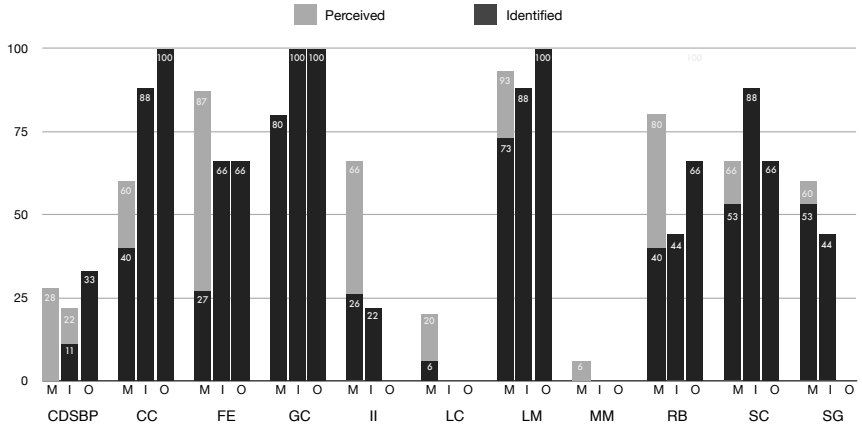


Figure 5.2: Percentage of (M)aster's students, (I)ndustrial developers, and (O)riginal developers that perceived and identified the bad smell examples in Eclipse.

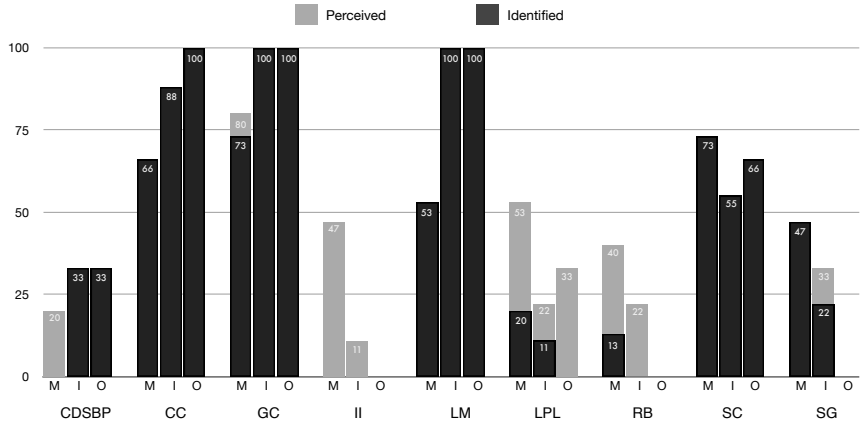


Figure 5.3: Percentage of (M)aster's students, (I)ndustrial developers, and (O)riginal developers that perceived and identified the bad smell examples in JEdit.

Before answering the two research questions formulated in Section 5.2.1, we analyze to what extent participants perceived a design problem in classes not containing any of the bad smells considered in our study. As previously explained, this is a sanity check aimed at verifying whether respondents were negatively biased. In total, we showed to participants 30 code components containing no smell (*i.e.*, 10 on `ArgoUML`, 11 on `Eclipse`, and 9 on `JEdit`). Master's students, industrial developers, and original developers marked as affected by design problems 10%, 5%, and 1%, respectively, of these code components. The low percentage indicates the absence of a negative bias in the respondents, and that this is particularly true for those with more experience (industrial developers) and with a better knowledge of the code (original developers). Moreover, when manually analyzing these cases of false positives, we found that most of the design problems observed by participants in classes not affected by any smell were related to problems in comments (*e.g., comments are missing*) or method/class naming (*e.g., class name does not reflect the class purpose*). In other words, in some sense the respondents correctly identified some kinds of problems, although these are not really structural code smells, but more similar to lexical smells [166], out of scope for this study.

Table 5.5: Median of the severity assigned by participants to the identified design problems.

System	Participants	CDSBP	CC	FE	GC	II	LC	LM	LPL	MM	RB	SC	SG
ArgoUML	Master's students	-	4	-	3	2	-	3	-	-	-	2	3
	Industrial	2	5	4	5	3	-	4	-	-	-	5	3
	Original	1	5	4	5	3	-	4	-	-	-	3	-
Eclipse	Master's students	-	4	4	5	4	1	4	-	-	4	3	3
	Industrial	3	4	5	5	4	-	5	-	-	-	5	3
	Original	3	5	5	5	-	-	5	-	-	4	5	-
JEdit	Master's students	-	4	-	3	-	-	3	3	-	4	3	3
	Industrial	2	5	4	5	3	-	4	3	-	-	5	3
	Original	2	5	-	5	-	-	5	-	-	-	4	-

Turning to the core of our study, Figures 5.1, 5.2, and 5.3, report the percentage of participants (of the different categories) that (i) perceived a problem in the analyzed code, and (ii) correctly identified the bad smell present in the code com-

ponent. Note that a code component that is correctly identified is also perceived (the opposite is not true). Columns labeled with "M", "I", and "O" report results for Master's students, industrial developers, and original developers, respectively. Also, Table 5.5 reports the median severity assigned by developers to the identified design/implementation problems[4].

5.3.1 Smells Generally not Perceived as Design or Implementation Problems

When looking at Figures 5.1, 5.2, and 5.3, one can immediately realize that some smells are, generally, not perceived as actual problems. This is particularly true for *Class Data Should Be Private*, *Middle Man*, *Long Parameter List*, *Lazy Class*, and *Inappropriate Intimacy*. In the following, we provide a qualitative analysis for them, based on the collected feedbacks and on the analysis of the code itself.

Class Data Should Be Private (CDSBP). This smell arises when a class exposes its attributes [8]. Respondents did not perceive this as an issue for the analyzed code components. Only a small percentage of Master's students perceived a problem in such components; however, they were never able to associate the problem to the characteristics of the CDSBP smell, mainly claiming issues related to poor commenting and methods complexity. Few (18% on average on the three systems) industrial developers recognized CDSBP as an issue, while one original developer for each system recognized the problem in the code. However, by looking at the severity values (Table 5.5), it emerges that respondents did not feel CDSBP as a real problem in the code. For instance, the severity assigned by original developers is *very low* (1) on `ArgoUML`, *medium* (3) on `Eclipse`, and *low* (2) on `JEdit`. It is interesting to report an observation made by the `ArgoUML` developer who recognized the CDSBP instance, while assigning it a *very low* severity: *"this class exposes all its fields, and this could look like bad coding. However, at the end of the day this is an*

[4]Complete data about the answers provided by participants are available in our replication package.

utility class with public static fields[5] *that can be used from everywhere in the system"*.

Middle Man (MM). *Middle Man* instances arise when a class is delegating all its work to other classes [8]. Classes affected by this smell were perceived by developers as classes without any design problem. The only exceptions are 13% and 6% of Master's students perceiving (but not identifying) a design problem on `ArgoUML` and `Eclipse`, respectively (note that this bad smell is not present in `JEdit`). Thus, high levels of delegation between classes do not seem to bother developers. In our understanding, developers could better perceive such a smell when doing performance analysis—*e.g.*, because the *Middle Man* could introduce overhead.

Long Parameter List (LPL). We found this smell in `JEdit` only (see Figure 5.3), where the `adjustDockingAreasToFit` method takes 11 parameters as input. While 53% of Master's students perceived a problem in the method, just 20% of them indicated the number of parameters as the issue. In most of the other cases, the problem felt by Master's students was the method complexity, the same perceived by the only original developer reporting a problem in the method. Finally, among the industrial developers, only one of them (11%) identified the problem in the method, however assigning it a severity of 3 (*medium*). The explanation provided justifies the low severity score: *"the method has several parameters; however, the feature implemented in it requires all of them"*. For this bad smell, the differences observed between the perception of students and professional developers can be explained as follows. Students are not used to large and complex projects. Consequently, they are more concerned by a method with several parameters as compared to more experienced developers. Also, original developers are aware of the reasons why such methods have a high number of parameters and are therefore not particularly concerned.

Lazy Class (LC). This smell represents a very small class that does little in the system. It is considered a bad smell since *"each class costs money to maintain and understand"* [8]. This smell affects only one of the investigated projects (`Eclipse`)

[5]Note that the fields were not *final*.

and it has been identified by only one Master's student (6%), that however ranked the problem as a *very low* severity one. In summary, the class of the `Eclipse` system named `SelectionOnNameOfMemberValuePair` was not considered as harmful by developers, since it just contains two methods, one of which is a simple `print` method. On the one hand, it is not surprising to see a lower severity perception here. On the other hand, in order to recognize a *Lazy Class* as possible problem, one should have clearly in mind whether having such an additional class could, in perspective, have benefits (*e.g.*, because the class itself is likely to evolve or to be extended by others), or negative effects (because it means scattering maintenance activities). Unfortunately, in this case we did not get opinions from original developers, the only ones that could have expressed an informed opinion.

Inappropriate Intimacy (II). This smell describes high levels of coupling between two classes [8]. Our results show that respondents did not consider high coupling as a problem. Master's students perceived a problem in the relationship between the two highly coupled classes in 48% of the cases, although they only identified the problem in 11% of the cases. Industrial developers were able to identify the problem in 18% of the cases while, among the original developers, only one of them recognized the existence of a coupling issue in `ArgoUML`, by assigning it a *medium* (3) severity. The two involved classes (*i.e.*, `ShortcutMgr` and `ActionWrapper`) have 27 dependencies among them. Despite that, the `ArgoUML` developer explained why this is not a big issue: "`ActionWrapper` *represents an action in the system that can be associated to a specific keyboard shortcut, while the class* `ShortcutMgr` *is in charge of managing all ArgoUML's shortcuts. Thus high coupling between these two classes is justified from my point of view*".

Summarizing, the bad smells described above are not perceived as problems by respondents that, consequently, are not able to identify them in source code. This is true for all the three categories of developers involved in our study, highlighting how developer's experience and system's knowledge do not play any important role in these cases. Also, it is interesting to observe that all these five smells (*i.e.*,

CDSBP, MM, LPL, LC, and II) are related to the lack of applying good Object-Oriented (OO) design practices, rather than to something one can easily perceive by looking at the code, (*i.e.*, as it would be for *God Class*). Indeed, by carefully looking at the definition of such smells, we notice that: CDSBP violates the information hiding principle; MM is a symptom of something wrong in the distribution of responsibilities between classes; LPL should be avoided in OO programming, since a method can ask other objects for the information it needs without the necessity of receiving all of them through parameters; LC often represents a class without a precise responsibility; and II identifies high-levels of coupling between classes.

5.3.2 Smells Generally Perceived and Identified by Respondents

There are some categories of smells that: (i) are highly perceived and identified by developers, (ii) create more concerns for developers having more experience and system knowledge, and (iii) are rated with high severity values. These smells are *Complex Class*, *God Class*, *Long Method*, and *Spaghetti Code*. As one can immediately notice, differently from the previous group, such smells are the ones for which the problem can be immediately perceived by looking at the code (which may be long and/or complex). In the following, we provide a detailed discussion for each of them.

Complex Class (CC). Original developers always identify this bad smell in the affected code components, assigning to it the maximum severity (*i.e.*, 5, *very high*). The provided explanations highlight the problems derived by classes having a high code complexity: *"the class is too complex, intricate, and very difficult to comprehend"*, *"several methods in this class are very complex, negatively affecting its maintainability"*. Also, industrial developers generally identify the problem (92% of cases, on average) while the less experienced participants (*i.e.*, Master's students) were able to describe the problem in 57% of cases, on average. Thus, higher experience seem to alert developers about problems caused by working on complex code. Note that the median of severity assigned by all participants to CC is *high* or *very high* (see Table 5.5).

God Class (GC). GC is the smell for which the respondents assigned the highest severity. Specifically, industrial and original developers always identified the problem in the analyzed code components, explaining how classes affected by GC are *"too large"*, *"a mixture of different responsibilities"*, resulting in *"difficulties in creating a mental model of how the class works"*. Industrial and original developers ranked GC with a median severity of 5 (*very high*). An example of GC instance evaluated in this study is the `GeneratorJava` class of `ArgoUML` composed by 66 methods and explicitly defined by one of the original developers as a *"big class in need of refactoring"*. Master's students were able to identify the design problem in 84% of cases, on average, by however assigning lower severity values than industrial and original developers (see Table 5.5). Thus, also on this bad smell higher developers experience seems to increase the threats perceived by GC instances.

Long Method (LM). Also this smell has been always identified by the original developers. The assigned median severity was 5 (*very high*) on `Eclipse` and `JEdit`, and 4 *high* on `ArgoUML`. An interesting comment made by one of the `ArgoUML` developers was *"this method is way too long, it could be split into three different methods"*. This comment confirms the *Long Method* bad smells as indicator of *Extract Method* refactoring [8] opportunities. Industrial developers identified the LM instances in 85% of cases, on average, assigning them a median severity of 4 (*high*) on `ArgoUML` and `JEdit`, and of 5 (*very high*) on `Eclipse`. On average, Master's students perceived the problem in 67% of the cases, correctly identifying it in 55% of the cases. Also in this case, the proportion of respondents with more experience that identified the problem is greater than the proportion of students.

Spaghetti Code (SC). On average, original developers identified SC instances in 77% of cases, followed by industrial developers (70%), and Master's students (61%). It is interesting to report the comment left by an `ArgoUML` developer: *"this class looks like procedural programming"*. This is exactly what the SC smell is: the abuse of procedural programming in OO code. The severity assigned by original and industrial developers is generally *high* or *very high*, compared to that perceived by students bounded between *low* and *medium*.

5.3.3 Smells whose Perception may Vary

Finally, there is a group of smells for which the perception varies case by case. Such smells are *Feature Envy*, *Refused Bequest*, and *Speculative Generality*.

Feature Envy (FE). This smell arises when a method is more interested in a class other than the one it is implemented in [8]. We found instances of FE in `ArgoUML` and `Eclipse`. Master's students almost always perceived some problems in methods affected by FE (100% in `ArgoUML` and 87% in `Eclipse`), however they always failed in correctly identifying the FE symptom in `ArgoUML`, and they only identify it in 27% of the cases in `Eclipse`. Instead, industrial developers were able to identify theFE instances in 50% of the cases, while for original developers this percentage goes up to 70%. One of the industrial developers explained that *"method* `parseMessage` *is placed in the wrong class. It should be moved to* `MyTokenizer` *since it is likely to change with that class"*. When identifying the FE smell, developers assigned to it high levels of severity, ranging from a median value of 4 (*high*) assigned by Master's students and industrial developers, to a median value of 5 (*very high*) assigned by original developers. As for other smells, it can be noticed that highly experienced developers are the ones that perceive this bad smell the most. We conjecture that FE smells are perceived mainly by original developers, because the "interest" of the FE method to other classes often grows over time and/or can be identified by the need for co-changing such a method together with other classes. Indeed, FE can be effectively identified by using historical data [52] (as explained later in Chapter 7).

Refused Bequest (RB). This smell arises between two classes when one inherits pieces of functionality from the other and never uses them. This is the only smell for which we have strong contradicting results across the three object systems. In particular, on `ArgoUML` (see Figure 5.1) and `JEdit` (see Figure 5.3) all respondents almost never perceived classes affected by RB as problematic ones. The situation is quite different on `Eclipse`, where 40% of Master's students, 44% of industrial developers, and 66% of original developers identified instances of RB in the analyzed pair of classes. We analyzed the instances of RB evaluated by participants

on the three object systems. What we found was that:

- in `ArgoUML`, the RB arises due to classes `TabSpawnable` and `TablePanel`. The latter overrides four out of the five methods inherited by `TabSpawnable`.
- in `JEdit`, the RB is between the class `CompletitionPopup` and the class `CompleteWord`. The latter overrides five out of the eight methods inherited by `CompletitionPopup`.
- in `Eclipse`, the RB is quite more extreme. In particular, the smelly class named `DefaultBindingResolver` inherits 53 methods from the parent class, which is `BindingResolver`, overriding 52 of them.

Note that in none of the three above cases the overriding methods invoke the `super` method of the superclass. The RB instance present in `Eclipse` was quite more visible than those present in `ArgoUML` and `JEdit`, concerning developers about its presence. The median severity assigned to this issue by original developers was *high* (4)–see Table 5.5.

Speculative Generality (SG). This smell represents the only one that was mainly perceived and identified by developers with low experience than by experienced ones. Master's students identified this smell in 53% of cases, on average, followed by industrial developers (33%), and original developers, never perceiving any design problem in classes affected by this smell. By looking back at the definition of SG, instances of this smell arise when a class is declared `abstract` but it is not specialized by any other class in the system. From the perspective of a Master's student, that mainly learned OO in courses and textbooks, this looks like a wrong usage of the OO paradigm. The median severity assigned by them is 3 (*medium*). The industrial developers identifying the problem (33% on average) also provided a *medium* severity to the SG instances, and one of them left a comment likely explaining the reason why those classes do not represent a problem for original developers: *"this class is* `abstract` *but not inherited by any class of the system. However, it could be that it is a partial implementation of something that will*

be integrated in the system in future system releases". In other words, without having a deep knowledge of the system, of the rationale behind the implementation choices, and of the project schedule, it could be difficult in some cases to assess design and/or implementation problems. Results obtained for this smell also warns against the abuse of too aggressive bad smell detectors that, in cases like this one, would report potential problems based on symptoms like the ones describing the SG. Only by observing the class evolution—*i.e.*, an `abstract` class would never be inherited during a long period of observation—one can say this is, indeed, a problem. Again, this reinforces the conjecture that historical data are extremely useful when identifying bad code smells [52].

5.4 Threats To Validity

Threats to *construct validity* are mainly related to how the sample of smells used in the study was identified, and to how we measured the developers' perception of code smells. Concerning the identification of the smell sample, a big threat can be due to the fact that, despite the presence of multiple evaluators minimized the possible effect of false positives, the identified smell instances may depend on the perception of who inspected the code to identify such smells. Hence, it could be the case that participants evaluated what the two students perceived as smells. However, the identification of smells was performed having all the possible support available, including smell definitions [8, 61], tools to compute metrics, and the source code change history. Certainly, in any case we had to limit to one (randomly selected) smell of each type per system, and this could have excluded instances of smells where the "magnitude" of the problem was more or less evident. However, such a kind of study involving industrial and original developers had quite strict constraints, *i.e.*, we could not afford to involve them in long inspection tasks on a huge number of smells.

Concerning the measure of perception (Section 5.2), we asked developers to tell us whether they perceived a problem in the code shown to them. In addition, we asked them to explain what kind of problem they perceived in order to un-

derstand whether or not they were able to correctly identify the design and/or implementation problem. Finally, for the severity we use a Likert scale [165] that allows to compare responses of multiple respondents. We are aware that questionnaires could only reflect a subjective perception of the problem, and might not fully capture the extent to which the bad smell could affect software development activities. To this aim, studies such as the one done by Yamashita and Moonen [9] are more suited.

Threats to *internal validity* may be related to factors that have influenced our results. One factor is the response rate: while appearing not very high (39%), it is higher than what it is normally expected in survey studies—i.e., below 20% [167]—even for the part of study done with the original developers, for which we obtained 22% return rate. Note also that we ensured a participation of at least three original developers for each system. We have limited a possible bias effect—*i.e.,* , the fact that developers could have told us that they perceived the presence of smells even in code not containing any smell—by also showing source code elements without smells.

Threats to *external validity* concern the generalization of our findings. Such threats can be related to (i) the set of chosen objects, (ii) the kinds of smells investigated in the study, and (iii) the pool of the participants of the study. Concerning the chosen objects, we are aware that our study is based on smells detected in three systems only, and that further studies are needed to confirm our conjecture. In this study we had to constrain our analysis to a limited set of smell instances, because the task to be performed by each respondent had to be reasonably small. In this study we covered a pretty large variety of smells, *i.e.,* the 12 described in Table 5.1. However, there are some smells we did not consider. Some of them, such as *Parallel Inheritance* or *Divergent Changes*, are smells which analysis explicitly requires a deep knowledge of the system history [52], and would therefore require a different type of study. Finally, for what concerns the participants, they represent different categories of developers, ranging from Master's students, representative of junior developers, to senior industrial programmers, and original developers of the investigated systems, having a deep knowledge of the code used in the study.

5.5 Conclusion

This chapter reported an empirical study aimed at analyzing to what extent bad code smells are perceived by developers as actual design and/or implementation problems. The study concerned examples of 12 kinds of smells detected in three Java open source projects—`ArgoUML`, `Eclipse`, and `JEdit`—and involved 10 original developers from the three projects and 24 outsiders, of which 9 are industrial developers and 15 are Master's students. The study results allowed us to distill the following lessons learned:

Lesson I. *There are some smells that are generally not perceived by developers as design problems*. Those smells are *Class Data Should Be Private*, *Middle Man*, *Long Parameter List*, *Lazy Class*, and *Inappropriate Intimacy*. As explained, these smells are all related to object-oriented (OO) good programming practice more than to complex/long code. Some of the explanations provided by developers highlighted as apparent violations of OO design principles, such as high levels of coupling or the absence of information hiding, do not necessarily reflect problematic situations. Sometimes they are simply the result of conscious choices made by developers. This underlines how approaches to (semi)automatically improve source code quality (*e.g.,* refactoring recommendation systems) cannot simply be evaluated through quality metrics, but should always be assessed with developers, in order to verify if the refactoring recommendations really reflect design problems from a developer's point-of-view.

Lesson II. *Instances of a bad smell may or may not represent a problem based on the "intensity" of the problem*. This, for example, happens for *Refused Bequest*, for which only the instance detected in Eclipse was recognized as a serious problem. This is pretty much consistent with results of the previous study made by Ratiu *et al.* for God classes [12]. This result highlights the usefulness of smell detectors providing a measure of severity [23].

Lesson III. *Smells related to complex/long source code are generally perceived as an important threat by developers*. This happens for *Complex Class*, *God Class*, *Long Method*,

and *Spaghetti Code*. Not only these smells were consistently identified by a very high proportion of respondents, but also they were rated with the highest level of severity. Intuitively, it could simply be the case that such smells are the easiest to be identified by developers, but it can also be that these are the problems for which developers require the most effective solutions, *i.e.,* precise recommenders that identify the smells and propose working solutions aimed at factoring them out.

Lesson IV. *Developer's experience and system's knowledge play an important role in the identification of some smells*. This is particularly true not only for the smells related to complex/long code, but also for smells related to possible mis-uses of OO principles, *e.g.,* the *Feature Envy*. This confirms that code quality assessment is a crucial task and team managers should allocate senior developers on them rather than junior programmers; despite a good academic background, the latter might not be able to properly identify and judge the problems in the code. In addition to that, as discussed above, an appropriate judgement of the severity of smells often require a good knowledge of the overall system design, of the rationale of decisions taken in the past, and of possible evolution trajectories the system would have in the future. Only experienced developers would know all these details.

As it always happens for empirical studies, the only way to corroborate our findings is to extend the study using a larger set of smells, other software, and different participants. Such replications are part of the agenda of our future work.

Chapter 6

An Experimental Investigation on the Innate Relationship between Quality and Refactoring

6.1 Introduction

Refactoring has been defined by Fowler as "*the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure*" [8]. This definition entails a strong relationship between refactoring and internal software quality, *i.e.,* refactoring improves software quality (*improves the software internal structure*). This has motivated research on bad smell and antipattern detection and on the identification of refactoring opportunities [25, 20, 52, 66, 27, 30, 34, 32].

However, whether refactoring is actually guided by poor design has not been empirically evaluated enough. Thus, this assumption still remains, for some aspects, a common wisdom that has generated controversial positions [121]. Specifically, there are no studies that **quantitatively** analyze which are the quality characteristics of the source code increasing their likelihood of being subject of refactoring operations. To the best of our knowledge, the available empirical evidence is based on two surveys performed with developers trying to understand the rea-

sons why developers perform refactoring operations [121, 120].

In addition, concerning the improvement of the internal quality of software, empirical studies have only shown that generally refactoring operations improve the values of quality metrics [168, 169, 170, 171], while the effectiveness of refactoring in removing design flaws (such as code smells) is still unknown.

In order to fill this gap, we use an existing tool, namely REF-FINDER [172], to automatically detect refactoring operations of 52 different types on 63 releases of three Java software systems, namely `Apache Ant`[1], `ArgoUML`[2], and `Xerces-J`[3]. Since REF-FINDER can identify some false positives, we manually analyzed the 15,008 refactoring operations detected by the tool. Among them, 2,086 were classified as false positives. Thus, in the context of our study we analyzed 12,922 refactoring operations.

Having identified the refactoring operations, for each class in the analyzed systems' releases we (i) measured a set of eleven quality metrics, and (ii) detected if it is affected by any instance of eleven code smells. Using these data we verify whether refactoring operations occur on code components for which the factors above (*i.e.,* quality metrics, presence of code smells) suggest there might be need for refactoring operations. In addition, we also measure the effectiveness of refactoring operations in terms of their ability to remove code smells.

The results achieved can be summarized as follows:

1. More often than not, quality metrics do not show a clear relationship with refactoring. In other words quality metrics might suggest classes as good candidates to be refactored that are generally not involved in developers' refactoring operations.

2. Among the 12,922 refactoring operations analyzed, 5,425 are performed by developers on code smells (42%). However, of these 5,425 only 933 actually remove the code smell from the affected class (7% of total operations) and 895 are attributable to only four code smells (*i.e.*, *Blob*, *Long Method*, *Spaghetti*

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

Table 6.1: Characteristics of the analyzed projects

Project	Period	Analyzed	#Releases	Classes	KLOC
Apache Ant	Jan 2000-Dec 2010	1.2-1.8.2	17	87-1,191	8-255
ArgoUML	Oct 2002-Dec 2011	0.12-0.34	13	777-1,519	362-918
Xerces-J	Nov 1999-Nov 2010	1.0.4-2.9.1	33	181-776	56-179
Overall	**-**	**-**	**63**	**-**	**-**

Code, and *Feature Envy*). Thus, not all code smells are likely to trigger refactoring activities.

In summary, such results suggest that (i) more often than not refactoring actions are not a direct consequence of worrisome metric profiles or of the presence of code smells, but rather driven by a general need for improving maintainability, and (ii) refactorings are mainly attributable to a subset of known smells. For all these reasons, the refactoring recommendation tools should not only base their suggestions on code characteristics, but they should consider the developer's point-of-view in order to propose meaningful suggestions of classes to be refactored.

6.2 Empirical Study Design

The *goal* of the study is to analyze refactoring operations occurring over the history of a software project, with the *purpose* of understanding (i) if quality metrics and code smells presence provide indications on which code components are more/-less likely of being refactored; and (ii) as a consequence, to what extent are refactoring operations effective in removing code smells from source code. The object systems, the tools, and the raw data are available for replication in our online appendix.[4]

6.2.1 Context and Research Questions

The study aims at addressing the following research questions:

[4] http://dx.doi.org/10.6084/m9.figshare.1207916

- **RQ1**: *Are refactoring operations performed on classes having a low-level of maintainability as indicated by quality metrics?*

- **RQ2**: *To what extent are refactoring operations (i) executed on classes exhibiting code smells and (ii) able to remove code smells?*

The *context* of the study consists of 63 releases of three Java open source projects, namely `Apache Ant`, `ArgoUML`, and `Xerces-J`. `Apache Ant` is a build tool and library specifically conceived for Java applications (though it can be used for other purposes). `ArgoUML` is an open source UML modeler, while `Xerces-J` is a XML parser for Java. Although this looks a relatively small context (three projects only), such a choice has been necessary to allow us manually validating the detected refactoring and code smells, as detailed below. Table 6.1 reports characteristics of the analyzed systems, namely analyzed releases, number of analyzed releases, and size range (in terms of KLOC and # of classes).

6.2.2 Study Variables and Data Extraction

The **dependent variables** considered in our study, for all the research questions, are the refactoring operations (of different types) being observed across releases of different programs. The **independent variables** are the factors we relate to such observed refactoring and namely:

1. For **RQ1**, a series of quality metrics (described below).

2. For **RQ2**, the presence of code smells (of different types) in software releases.

To answer our research questions, we first need to detect refactorings over the evolution history of the studied systems. To this aim we use an existing tool, REF-FINDER [172], to detect refactoring operations performed between each subsequent couples of releases of each system. REF-FINDER has been implemented as an Eclipse plug-in and it is able to detect 63 different kinds of refactoring operations. In a case study conducted on three open source systems, REF-FINDER was able to detect refactoring operations with an average recall of 95% and an average

Table 6.2: Refactoring operations analyzed

Project	#Refactorings	Distinct types of refactorings
Apache Ant	1,469	31
ArgoUML	3,532	43
Xerces-J	7,921	43
Overall	**12,922**	**52**

precision of 79% [172]. Even if the accuracy of such a tool is quite high, we tried to (at least) mitigate problems related to false positives (precision) through manual validation of the refactoring operations identified by REF-FINDER. Specifically, each refactoring operation identified by the tool was manually analyzed through source code inspection by two Master's students from the University of Salerno. The students individually validated each of the proposed refactoring operations.

Once students validated the refactoring operations, they performed an open discussion to solve conflicts and reach a consensus on the refactoring operations analyzed, classifying them as *true positive* or *false positive*. Of the 15,008 refactoring operations detected by Ref-Finder, 12,922 operations have been manually classified as actual refactoring operations, producing as output a set of triples (rel_j, ref_k, C), where rel_j indicates the release number, ref_k the kind of refactoring occurred, and C is the set of refactored classes.

Table 6.2 reports the number of refactoring operations (as well as the number of different types of refactorings) identified on the three systems after the manual validation. While the extracted refactoring operations are needed to answer all our research questions, in the following we detail on data collection activities made to specifically answer each research question.

Data Extracted to Answer RQ1

To answer **RQ1**, we need to measure—for each class of the analyzed systems—a set of quality metrics. Specifically, we measure for each class in the analyzed systems' releases a set of eleven quality metrics. Since we know in each release which classes have been subject of which refactoring operations, we can use these

Table 6.3: Quality metrics measured to answer **RQ1**

Metric	Description
Lines of Code (LOC)	The number of lines of code excluding white spaces and comments
Weighted Methods per Class (WMC) [64]	The sum of the McCabe's cyclomatic complexity of its methods
Depth of Inheritance Tree (DIT) [64]	The depth of a class as the number of its ancestor classes
Number Of Children (NOC) [64]	# of direct descendants (subclasses) of a class
Response for a Class (RFC) [64]	# of distinct methods and constructors invoked by a class
Coupling Between Object (CBO) [64]	# of classes to which a class is coupled
Lack of COhesion of Methods (LCOM) [64]	The higher the pairs of methods sharing at least a field, the higher the cohesion of the class
# of Operations Added by a subclass (NOA) [181]	# of methods added by a subclass to the methods inherited by its superclass
# of Operations Overridden by a subclass (NOO) [181]	# of methods overridden by a subclass among those inherited by its superclass
Conceptual Coupling Between Classes (CCBC) [182]	The higher the textual similarity between two classes, the higher their coupling
Conceptual Cohesion of Classes (C3) [70]	The higher the textual similarity between methods, the higher the cohesion of the class

metrics to understand if any of them suggest that the considered classes need to be refactored.

The employed quality metrics are reported in Table 6.3. Our choice of the metrics is not random. We considered LOC since it has been demonstrated to be one of the better metrics in predicting the number of faults in a code component [173]. Thus, it is also possible that LOC also helps in identifying classes having a poor design from the developers point of view. The Chidamber & Kemerer (CK) metrics [64] have been object of several empirical studies showing their ability of capturing different aspects of code maintainability [64, 174, 175, 176, 177, 178, 179]. We also adopted NOA and NOO since they measure quality aspects of a class that are not taken into account by the CK metrics (see Table 6.3). Finally, we also considered semantic metrics since (i) they have been shown to not correlate with structural metrics [70] and (ii) in a recent study [180] the Conceptual Coupling Between Classes (CCBC) has been shown to be the coupling metric better capturing the developers perception of coupling between code components. To extract these metrics, we developed a tool exploiting the Eclipse JDT API to extract all needed information from source code.

Data Extracted to Answer RQ2

To answer **RQ2**, we analyze each class of the 63 considered software releases to verify if it is affected by any code smell. In particular, we detected instances of

Table 6.4: Code smells detected to answer **RQ3**

Name	Description
Class Data Should Be Private (CDSBP) [61]	A class exposing its fields, violating the principle of data hiding.
Complex Class [61]	A class having at least one method having a high cyclomatic complexity.
Feature Envy [8]	A method is more interested in a class other than the one it actually is in.
Blob Class (Blob) [61]	A large class implementing different responsibilities and centralizing most of the system processing.
Lazy Class [8]	A class having very small dimension, few methods and with low complexity.
Long Method [8]	A method that is unduly long in terms of lines of code.
Long Parameter List (LPL) [8]	A method having a long list of parameters, some of which avoidable.
Message Chain [8]	A long chain of method invocations is performed to implement a class functionality.
Refused Bequest [8]	A class redefining most of the inherited methods, thus signaling a wrong hierarchy.
Spaghetti Code [61]	A class implementing complex methods interacting between them, with no parameters.
Speculative Generality [8]	A class declared as abstract having very few children classes using its methods.

the eleven code smells reported in Table 6.4 defined by Fowler [8] and Brown *et al.* [61]. Also in this case, the goal is to understand if the presence of specific code smells increases/decreases the changes of the affected code components of being the object of refactoring actions. To detect the code smells we developed a simple tool that outputs a list of candidate classes potentially exhibiting a code smell. Then, we manually validated the candidate code smells suggested by the tool. The validation was performed by a Master and a Ph.D. student, who individually analyzed and classified as *true positive* or *false positive* all candidate code smells. Finally, the students performed an open discussion with researchers to resolve any conflicts and reach a consensus on the detected code smells.

To ensure high recall, our tool uses very simple detection rules that overestimate the presence of code smells in the code. This is done at the expense of precision. Even though this choice resulted in a longer list of candidates and thus in a more expensive manual validation, it was necessary to study the real distribution of code smells in the analyzed releases. Table 6.5 reports the rules applied by our tool to detect each of the eleven analyzed code smells. Note that we choose not to use existing detection tools [25, 20, 66] because (i) none of them has ever been applied to detect all the studied code smells, and (ii) their detection rules are generally restrictive to ensure a good compromise between recall and precision, thus they may miss some code smell instances. Table 6.6 reports the number of code smells and the number of different types of smells identified on the three systems after the manual validation.

Table 6.5: The rules used by our tool to detect candidate code smells

Name	Description
Class Data Should Be Private	A class having at least one public field.
Complex Class	A class having at least one method for which McCabe cyclomatic complexity is higher than 10.
Feature Envy	All methods having more calls with another class than the one they are implemented.
Blob Class	All classes having (i) cohesion lower than the average of the system AND (ii) LOCs $>$ 500.
Lazy Class	All classes having LOCs lower than the first quartile of the distribution of LOCs for all system's classes.
Long Method	All methods having LOCs higher than the average of the system.
Long Parameter List	All methods having a number of parameters higher than the average of the system.
Message Chain	All chains of methods' calls longer than three.
Refused Bequest	All classes overriding more than half of the methods inherited by a superclass.
Spaghetti Code	A class implementing at least two long methods interacting between them through method calls or shared fields.
Speculative Generality	A class declared as abstract having less than three children classes using its methods.

Table 6.6: Number of smells analyzed

Project	#Smells	Distinct types of smells
Apache Ant	1,493	10
ArgoUML	1,197	7
Xerces-J	2,788	10
Overall	**5,478**	**10**

Knowing the list of classes affected by each code smell in each software release, we are also able to verify to what extent refactoring operations are able to remove code smells from source code. In particular, given a refactoring operation (*e.g.*, Extract Class) o_i performed in a release r_j on a class affected by a code smell (*e.g.*, Blob class) a_k, we can verify if o_i was able to remove a_k by checking if a_k is still present in the release r_{j+1} (and thus, the code smell has not be removed) or not (the code smell has been removed).

6.2.3 Analysis Method

To address the two research questions formulated above, we build, for each object system and for each kind of refactoring operation performed on it, logistic regression models relating a (dichotomous) dependent variable—indicating whether or not a particular type of refactoring was performed—with independent variables represented by the quality indicators (metrics, and presence of code smells) described above. Logistic regression models [162] relate dichotomous dependent

variables with one or more independent variables as follows:

$$\pi(X_1, X_2, \ldots, X_n) = \frac{e^{C_0+C_1\cdot X_1+\cdots+C_n\cdot X_n}}{1+e^{C_0+C_1\cdot X_1+\cdots+C_n\cdot X_n}} \tag{6.1}$$

where X_i are the independent variables describing the phenomenon, and C_i the coefficients (estimates) of the logistic regression model. We used the R statistical software (`http://www.r-project.org/`) to build the logistic regression models. Specifically, we built the following two models:

1. *Metrics*. The first model uses the eleven measured quality metrics as independent variables and the application of the particular type of refactoring (*e.g.,* Extract class) as the dependent variable. All metrics have been normalized using the z-score, *i.e.,* by subtracting the mean and dividing by the standard deviation.

2. *Smells*. The second model uses the presence of the considered code smells in a class as independent (and boolean) variables, and the application of the particular type of refactoring (*e.g.,* Extract class) as the dependent variable.

Note that, given a refactoring type r_i and a system s_j, we build the two models presented above only if the refactoring type r_i has been applied on the system s_j at least 10 times. This is done to avoid the creation of unreliable logistic regression models.

We are aware that our models could be affected by multi-collinearity [183], which occurs when two or more independent variables are highly correlated and can be predicted one from the other, possibly affecting the resulting model. We assess our models for the presence of multi-collinearity in two different ways:

1. Whenever possible, *i.e.,* for the models based on metrics, we compute the Spearman's rank correlation [184] between all possible pairs of metrics, to determine whether there are pairs of strongly correlated metrics (*i.e.,* with a Spearman's $\alpha > 0.8$). If two independent variables are highly correlated, one of them should be removed from the model.

2. By using a stepwise variable removal procedure based on the Companion Applied Regression (*car*) R package[5], and in particular based on the *vif* (variance inflation factors) function [183].

Once we have avoided multi-collinearity using the procedure described above, we build the logistic regression models with the variables remained after the pruning. Then, for each model we analyze (i) whether each independent variable is significantly correlated with the dependent variable (we consider a significance level of $\alpha = 5\%$), and (ii) we quantify such a correlation using the Odds Ratio (OR) [185] which, for a logistic regression model, is given by e^{C_i}. The higher the OR for an independent variable, the higher its ability to explain the dependent variable. However, the interpretation of the OR changes between the two kinds of models we built, due to the different measurement scale of the independent variables, *i.e.*, ratio for the metric-based model and nominal (categorical) for the code smell-based model. In particular, for the model built using quality metrics, the OR for an independent variable indicates the increment of chances for a class to be subject of refactoring in consequent of a one-unit increase of the independent variable. For example, if we found that the CBO has an OR of 1.15 when building a logistic regression model for the Extract Class refactoring operation, this means that for each one-unit increase of the CBO value for a class, it has 15% higher chances of being involved in an Extract Class refactoring operation. On the other side, for the model built using code smells, the OR indicates the likelihood of a class affected by a code smell of being involved in refactoring operations with respect to a non-affected class. As example, if we found that the code smell *Blob* has an OR of 3 when building a logistic regression model for the Extract Class refactoring operation, this means that classes affected by the *Blob* code smell have 3 times higher chances of being involved in an Extract Class refactoring operation than classes not affected by it.

Finally, to verify the ability of refactoring in removing code smells from source code, we simply analyze for each refactoring type (*e.g.*, Extract class) the percent-

[5]http://cran.r-project.org/web/packages/car/index.html

age of times it is able to remove each type of code smell (*e.g., Blob* class).

6.3 Empirical Study Results

This section discusses the results of our study, aimed at addressing the research questions formulated in Section 9.3.1. As explained in Section 6.2.3, before building the logistic regression models, we performed a multi-collinearity analysis. As a result of such analysis, we found that:

- For the models based on metrics, and only for the Xerces project, the stepwise regression procedure removed the DIT metric from the logistic regression model. Consistently with that, we found a strong ($\alpha = 0.83$) Spearman's rank correlation between DIT and NOA. This is not entirely surprising as both DIT and NOA capture information related to inheritance relations between classes. No multi-collinearity was found for the other two projects (Apache Ant and ArgoUML).
- For the models based on smells, no independent variable is affected by multi-collinearity.

6.3.1 Are refactoring operations performed on classes having a low-level of maintainability as indicated by quality metrics?

Table 6.7 reports the ORs of quality metrics obtained when building a logistic regression model for data concerning each refactoring operation. Statistically significant values, i.e., those for which the p-value is lower than 0.05, are reported in **bold** face. In the following, we will mainly focus our discussion on such statistically significant values.

First, we can immediately notice that longer classes (in terms of LOC) generally have a higher chance of being involved in a refactoring operations (the ORs for LOC are higher than 1 in 71% of significant ORs). This is quite an expected result.

Table 6.7: Quality metrics model: OR of metrics when building logistic model. Statistically significant ORs are reported in bold face.

Refactoring	System	LOC	WMC	DIT	NOC	RFC	CBO	LCOM	NOA	NOO	CCBC	C3
add parameter	ApacheAnt	**3.51**	0.40	**0.39**	**1.24**	1.61	**0.41**	1.26	0.59	0.98	0.75	**0.22**
add parameter	ArgoUML	**1.15**	**1.01**	1.62	1.09	1.45	**0.79**	**0.04**	**1.04**	0.69	0.34	0.54
add parameter	Xerces	1.06	**2.26**	-	**1.13**	**0.67**	**1.10**	**0.66**	1.31	0.99	0.30	**0.56**
consolidate cond expression	ApacheAnt	0.27	**22.35**	**1.00**	**0.90**	**1.58**	**0.36**	**0.23**	0.14	1.04	1.54	**0.34**
consolidate cond expression	ArgoUML	1.79	1.07	1.21	**0.77**	**3.05**	**0.74**	**0.01**	**1.44**	1.28	1.56	0.30
consolidate cond expression	Xerces	1.30	9.54	-	1.07	0.63	1.01	0.78	1.10	1.13	1.49	0.36
consolidate duplicate cond fragments	ApacheAnt	0.53	8.02	0.53	**1.25**	**2.35**	**0.38**	0.47	**0.54**	**0.63**	0.91	**0.31**
consolidate duplicate cond fragments	ArgoUML	**1.09**	**1.92**	**1.23**	**1.02**	**2.62**	**1.35**	0.00	0.53	1.29	0.54	**0.72**
consolidate duplicate cond fragments	Xerces	1.26	6.77	-	**1.13**	**0.77**	**1.86**	0.92	1.13	1.02	0.68	0.56
extract method	ApacheAnt	**0.83**	5.84	0.40	1.24	**2.42**	0.58	**0.31**	**0.34**	0.84	0.78	0.20
extract method	ArgoUML	**1.29**	**0.27**	1.55	1.03	2.40	**1.12**	**0.05**	0.94	0.86	0.35	0.28
extract method	Xerces	**1.16**	0.64	-	**1.11**	1.33	**1.49**	0.66	0.99	1.08	0.67	**0.28**
extract superclass	ArgoUML	**2.56**	**0.36**	**0.45**	**1.11**	**0.65**	1.04	0.00	**0.68**	**0.38**	0.95	**0.06**
form template method	ArgoUML	**4.10**	**0.00**	0.00	1.82	**0.88**	1.18	**0.00**	**2.94**	**1.82**	0.00	**0.38**
inline method	ApacheAnt	1.16	**0.14**	3.23	1.41	**4.04**	**0.20**	0.11	0.13	0.42	1.40	0.08
inline method	ArgoUML	1.15	0.63	3.46	1.15	1.46	1.35	0.24	0.84	0.85	0.17	0.30
inline method	Xerces	0.71	1.59	-	**1.10**	**1.10**	**1.15**	**0.39**	0.98	**1.30**	**1.14**	**0.07**
inline temp	ApacheAnt	**1.56**	3.55	0.57	1.14	**0.59**	0.98	**0.85**	**0.35**	0.63	**0.95**	**0.29**
inline temp	ArgoUML	1.17	0.96	1.58	0.43	**1.02**	**1.31**	**0.16**	**0.78**	0.94	**0.82**	0.29
inline temp	Xerces	1.80	**9.94**	-	1.09	**0.61**	1.64	**0.68**	1.37	1.00	**0.97**	**0.70**
introduce assertion	ArgoUML	0.13	0.68	6.97	0.68	6.23	1.83	0.02	0.27	0.00	0.83	**0.39**
introduce explaining variable	ApacheAnt	1.54	**0.81**	1.14	**1.29**	1.88	0.61	1.04	0.10	0.18	1.04	0.16
introduce explaining variable	ArgoUML	0.82	1.05	**0.83**	1.00	2.54	1.16	0.27	0.80	0.99	**0.69**	0.53
introduce explaining variable	Xerces	1.00	4.12	-	1.11	0.86	**1.81**	**0.97**	1.04	1.02	0.80	**0.46**
introduce null object	ArgoUML	0.42	1.90	**1.12**	0.97	**0.00**	**2.21**	0.00	4.91	2.53	1.52	0.92
introduce parameter object	Xerces	0.88	2.97	-	1.08	**1.33**	0.25	0.15	1.36	0.91	0.00	**0.06**
move field	ApacheAnt	7.53	0.02	2.80	1.35	**6.82**	0.12	1.51	0.43	0.47	0.65	0.23
move field	ArgoUML	**10.40**	**0.00**	**1.58**	0.92	0.77	1.16	**0.00**	**0.75**	**0.24**	1.08	**1.72**
move field	Xerces	1.07	2.63	-	1.00	**0.61**	**1.58**	**0.89**	**1.04**	1.10	**0.55**	0.10
move method	ApacheAnt	1.41	0.10	0.51	1.39	**7.13**	0.25	**0.77**	1.04	0.37	**1.13**	**0.83**
move method	ArgoUML	**1.18**	**1.61**	**2.97**	1.06	**0.60**	1.26	**0.04**	**0.81**	**0.92**	**0.58**	**1.22**
move method	Xerces	1.03	2.91	-	**0.82**	**0.50**	1.29	**0.71**	1.18	1.12	**0.50**	**0.13**
pull up field	Xerces	**2.44**	0.37	-	**1.17**	0.89	**1.90**	**0.60**	20.31	6.07	**0.25**	**0.73**
pull up method	Xerces	1.71	0.00	-	0.03	0.00	4.20	0.00	8.26	20.91	**0.45**	0.00
push down field	Xerces	**5.49**	**0.22**	-	2.34	**0.06**	**0.86**	0.44	0.31	**1.64**	**0.23**	**1.43**
push down method	Xerces	29.65	0.00	-	1.54	0.00	**3.00**	**0.07**	**0.32**	1.42	**0.38**	**1.18**
remove assignment to parameters	ApacheAnt	0.25	4.35	0.73	1.02	**0.37**	0.85	0.31	0.38	0.00	1.09	1.19
remove assignment to parameters	ArgoUML	1.52	**0.29**	1.24	**0.34**	**1.88**	0.82	**0.00**	**1.12**	0.65	**0.22**	**0.25**
remove assignment to parameters	Xerces	1.73	**0.34**	-	1.10	**1.29**	0.99	**1.01**	1.26	**0.85**	**0.47**	**0.58**
remove control flag	ApacheAnt	**0.23**	**5.47**	**0.32**	**0.13**	**1.06**	**0.16**	0.24	0.26	0.69	**1.59**	**0.46**
remove control flag	ArgoUML	1.38	2.19	1.47	**0.97**	**1.65**	**1.06**	**0.10**	1.53	0.37	**0.69**	**0.22**
remove control flag	Xerces	2.32	1.68	-	0.85	0.66	1.15	**0.75**	0.92	**0.91**	0.37	**0.39**
remove parameter	ApacheAnt	**2.26**	**0.77**	0.51	1.28	**1.08**	**0.51**	**1.13**	**0.55**	0.59	0.80	0.22
remove parameter	ArgoUML	1.10	**0.93**	1.16	**1.12**	1.42	**0.95**	0.06	0.88	0.89	0.36	**0.59**
remove parameter	Xerces	0.96	1.41	-	1.12	1.06	**0.87**	**0.68**	1.19	0.99	**0.36**	0.35
rename method	ApacheAnt	10.76	**0.00**	**3.08**	1.73	**9.88**	**0.13**	**0.95**	**0.07**	**0.08**	**1.03**	**0.03**
rename method	ArgoUML	1.29	**0.90**	**1.01**	**1.13**	**0.98**	**1.22**	**0.12**	**1.10**	**0.96**	**0.34**	**0.26**
rename method	Xerces	0.62	8.61	-	1.05	0.35	0.83	0.64	1.27	1.30	0.77	0.09
replace data with object	ArgoUML	**0.38**	**0.95**	2.38	1.19	1.20	**1.57**	0.12	1.02	**0.19**	**0.01**	0.39
replace data with object	Xerces	**1.81**	1.47	-	**0.96**	**0.36**	1.32	**1.40**	**1.26**	**1.31**	**0.24**	0.15
replace exception with test	Xerces	**8.43**	**0.74**	-	**0.00**	**0.48**	0.03	**1.86**	7.48	0.00	0.09	0.42
replace magic number with constant	ApacheAnt	**0.37**	10.04	0.86	0.55	0.51	1.15	0.57	0.02	0.26	**0.81**	0.57
replace magic number with constant	ArgoUML	**1.95**	0.04	1.87	0.28	**2.18**	0.56	**0.76**	0.77	0.57	**0.12**	**0.26**
replace magic number with constant	Xerces	0.77	2.64	-	**0.92**	0.61	**3.63**	**1.03**	**1.05**	0.90	**0.54**	**0.51**
replace method with method object	ApacheAnt	**0.61**	**5.84**	**0.39**	0.12	**4.51**	**0.27**	**1.05**	**0.45**	**1.08**	**0.31**	**0.40**
replace method with method object	ArgoUML	1.21	0.80	2.16	0.97	**1.17**	1.68	0.10	**0.74**	**0.91**	**0.77**	**0.68**
replace method with method object	Xerces	1.69	0.92	-	1.04	0.80	**1.15**	1.08	0.98	1.04	**0.47**	0.24
replace nested cond guard clauses	ApacheAnt	**0.22**	**8.92**	**1.31**	0.42	**1.11**	**0.87**	0.09	0.93	**0.62**	**0.61**	**1.26**
replace nested cond guard clauses	ArgoUML	**0.66**	3.86	**0.44**	0.98	2.50	0.87	0.01	1.48	0.86	**1.03**	**0.55**
replace nested cond guard clauses	Xerces	**1.64**	1.56	-	1.09	0.94	**1.42**	0.87	0.96	1.06	0.76	**0.19**
separate query from modifier	Xerces	0.76	3.31	-	1.04	0.42	1.63	0.60	1.33	0.54	**3.56**	0.95

More interesting are the results—and in particular the observed OR values—for the other metrics.

The WMC metric of a class, *i.e.*, the sum of the McCabes' cyclomatic complexity of its methods, exhibits very high ORs for some of the refactoring operations dealing with the simplification of methods inside a class. However, this is not always true for all systems. In particular, classes having high WMC have:

- In `Apache Ant` (OR 22.35), a much higher chance of being involved in a *consolidate conditional expression* refactoring, performed to simplifying a sequence of conditional expressions which produce the same result by combining them into a single expression. The OR for WMC on this refactoring is also very high on `ArgoUML` (9.54), even if not statistically significant.

- In `Apache Ant` (OR 5.47), a higher chance of being involved in a *remove control flag* refactoring, performed to replace a variable that is acting as a control flag for a series of boolean expressions with a simpler break statement. In this case, also on the other systems the OR is higher than 1, but not statistically significant.

- In `Apache Ant` (OR 8.9), a higher chance of being involved in a *replace nested conditional with guard clauses* refactoring, applied to methods in which the conditional behavior does not make clear the normal path of execution. Also in this case, on both other systems the OR is higher than 1, but not statistically significant.

- In `Xerces` (OR 9.94), a higher chance of being involved in an *inline temp* refactoring, performed to remove temporary variables that are only assigned once with a simple expression. Also in `Apache Ant` the OR for this refactoring is high (3.55) but, again, not statistically significant.

Surprisingly, we did not find any statistically significant OR higher than one for WMC on models built for the *extract method* refactoring (see Table 6.7).

Concerning DIT, the metric measuring the depth of a class as the number of its ancestor classes, we expect strong ORs for refactoring operations dealing with

changes applied to the class hierarchy (*i.e., push down method*, *pull up method*, *pull up field*, *push down field*, *form template method*, and *extract superclass*). However, we do not observe any statistically significant OR higher than one.

As for the NOC metric, counting the number of direct descendants (subclasses) of a class, we expected high ORs for refactoring operations acting on the class hierarchy. However, the ORs we found are either not statistically significant, or very close to 1 (see Table 6.7).

NOA (the number of operations added by a subclass) and NOO (the number of operations overridden by a subclass) are also related to class hierarchies and, in such cases, results confirm the conjecture that such metrics can relate with refactorings. Firstly, both metrics show high ORs with the *form template method* refactoring, which is often applied when in two subclasses there are very similar methods. These two methods are generally merged into a single one that is pulled up in the class hierarchy. For this reason, NOA and NOO also exhibit very high ORs with the *pull up method* and *pull up field* refactorings, even if these are not statistically significant.

RFC measures the coupling of a class and thus, we expect it to obtain high ORs for refactoring operations allowing a coupling reduction (*e.g.,* inline method, move method, move field). Concerning the *inline method* refactoring, applied to merge two very coupled methods, we found ORs higher than 1 for all object systems, showing that highly coupled classes have a higher chance of being involved in such refactoring. However, for operations like *move method* and *move field*, we found contradicting results. Specifically, for these two refactorings we found very high ORs on `Apache Ant` (7.13 for *move method* and 6.82 for *move field*) together with ORs lower than 1 on the other two systems. We also found very high ORs for other refactoring operations that, however, do not allow to reduce coupling (see *e.g., rename method* with an OR of 9.88 in `Apache Ant`).

CBO, also related to coupling, mainly exhibits high ORs for refactoring operations that are not related to a coupling reduction (*e.g., replace method with method object* with an OR of 3.63 in `Xerces`). The only expected result we found is that classes having high CBO (and thus, having several dependencies with other classes)

have a higher chance of being involved in a *push down method* refactoring (OR equals 3.00) and generally have a higher chance of being involved in all refactoring operations moving code components among the class hierarchy. This result is expected since classes having a high CBO are also more likely to have inheritance dependencies with other classes. In fact, the CBO counts the number of objects with which a class has dependencies, including inheritances.

Table 6.8: Quality metrics model: summary of results.

Metric	Refactoring operations related to the metric		Overlap
	Expected	Found	
LOC	All	add parameter; extract superclass; form template method; inline temp move field; move method; pull up field; pull up method; push down field; push down method; replace exception with test	39%
WMC	add parameter; consolidate cond expression; consolidate duplicate cond fragments; extract method; remove control flag; replace nested cond guard clauses	consolidate cond expression; consolidate duplicate cond fragments; remove control flag; replace nested cond guard clauses	67%
DIT	pull up method; pull up field; push down field; push down method; form template method; extract superclass	introduce null object	0%
NOC	pull up method; pull up field; push down field; push down method; form template method; extract superclass	add parameter; extract superclass; consolidate duplicate cond fragments; introduce explaining variable; pull up field; remove parameter	20%
RFC	inline method; move field; move method	extract method; inline method; remove parameter	20%
CBO	inline method; move field; move method; pull up method; pull up field; push down field; push down method; form template method	introduce null object; pull up field; push down method; replace data with object	20%
LCOM	move field; move method	replace exception with test	0%
NOA	pull up method; pull up field; push down field; push down method; form template method; extract superclass	form template method	17%
NOO	pull up method; pull up field; push down field; push down method; form template method; extract superclass	form template method; push down field	30%
CCBC	inline method; move field; move method; pull up method; pull up field; push down field; push down method; form template method; rename method	separate query from modifier	0%
C3	move field; move method; rename method	push down field; push down method	0%

The structural cohesion metric LCOM does not provide any interesting result, generally showing low OR for the different refactoring operations. Some interesting results were achieved for the semantic cohesion metric C3, for which we observed an OR higher than 1 for *move method* and *move field* refactoring on `ArgoUML`. This indicates that some responsibilities of classes having low C3 (conceptual cohesion) are extracted from such classes. Finally, concerning the semantic coupling

metric CCBC, it shows a high OR for the *separate query from modifier* refactoring. However, this refactoring operation does not deal with coupling reduction. While in some cases ORs higher than 1 are obtained for refactoring reducing coupling (*e.g., move method* on `Apache Ant`), as already observed for the structural coupling metric RFC, this result is not confirmed on all the other systems, exhibiting ORs lower than 1.

Table 6.8 summarizes the results achieved for the quality metrics model by reporting for each of the investigated metrics:

1. The refactoring operations for which we expected some form of correlation. For example, we expect that classes having a high WMC value (WMC measures the code complexity) are more subject to refactoring operations aiming at reducing code complexity like, for example, *extract method*.

2. The refactoring operations for which we observed evidence of a relationship with quality metrics profile. In this case we mean refactoring operations for which we observed (i) a statistically significant OR higher than one for at least one of the object systems and (ii) consistent results (*i.e.*, OR higher than one, even if not statistically significant) on the other systems.

3. The percentage of overlap between the set of expected refactorings (point 1) and the set of refactorings for which we actually observed some form of correlation (point 2).

Table 6.9: Code smells identified in each system (among all analyzed versions).

System	Blob	CDSBP	Complex Class	Lazy Class	Long Method	LPL	Message Chain	Refused Bequest	Spaghetti Code	Speculative Generality	Feature Envy
ApacheAnt	85	370	0	167	110	12	0	5	9	40	62
ArgoUML	196	343	67	351	151	31	0	56	28	185	291
Xerces	328	792	48	664	700	17	0	852	71	124	34

The analysis of Table 6.8 highlights that *with very few exceptions, quality metrics do not show a clear relationship with refactoring*. The only exception is represented

by the WMC metric, that seems to be able to indicate classes attracting the developers' refactoring attentions. As for the other metrics, none of them showed with strong evidence relation with refactoring. Particularly surprising are the results achieved with cohesion and coupling metrics, generally considered good indicators of source code components in need of refactoring [27]. It is important to point out that we are not claiming the opposite being generally true, but just reporting that *refactoring operations do not target classes exhibiting low cohesion and/or high coupling as much as expected*.

6.3.2 To what extent are refactoring operations (i) executed on classes exhibiting code smells and (ii) able to remove code smells?

Table 6.9 reports the number of classes affected by the different code smells we identified in the analyzed releases. Note that, for each system, we report the overall number of code smells identified across all the analyzed releases. This means that if a class is affected by a code smell in all the 33 analyzed Xerces releases, this class has been counted 33 times. We did not find any Message Chain code smell. Thus, we will not discuss it in the following results analysis.

Table 6.10 reports the ORs obtained for the considered code smells when building a logistic regression model for data concerning each refactoring operation (as explained in Section 6.2). Moreover, we also show in Table 6.11, the number of refactorings performed on each type of code smell, and in Table 6.12 the percentage of code smells removed when developers performed refactoring actions.

The analysis of ORs reported in Table 6.10 highlights that Blob classes are generally subject to refactoring. A Blob is a large class implementing different responsibilities and centralizing most of the system behavior. Note that this is somewhat an expected result, and consistent with the findings related to the metric model (Table 6.8). Indeed, Blob classes are quite large in terms of LOCs and, as observed while discussing the quality metrics results, larger classes generally have a higher chance of being involved in a refactoring operation. This result is also confirmed

Table 6.10: Code smell model: OR of smells when building logistic regression model. Statistically significant ORs are reported in **bold face**.

Refactoring	System	Blob	CDSBP	Complex Class	Lazy Class	Long Method	LPL	Refused Bequest	Spaghetti Code	Speculative Generality	Feature Envy
add parameter	ApacheAnt	**6.53**	1.74	0.00	0.00	**3.25**	0.00	0.00	**7.40**	0.00	0.00
add parameter	ArgoUML	**2.66**	**1.83**	0.00	**0.12**	**3.62**	0.00	0.85	0.00	0.86	1.02
add parameter	Xerces	1.10	**0.70**	0.39	0.00	**2.27**	**2.70**	**0.11**	**2.35**	**4.14**	0.75
consolidate cond expression	ApacheAnt	**9.33**	0.46	0.00	0.00	0.00	0.00	0.00	195.80	0.00	0.00
consolidate cond expression	ArgoUML	3.29	0.90	0.00	0.00	**5.16**	0.00	0.00	0.00	0.00	**3.71**
consolidate cond expression	Xerces	**2.45**	0.91	0.00	0.00	**1.79**	**5.44**	0.37	0.61	1.98	0.00
consolidate duplicate cond fragments	ApacheAnt	**2.72**	1.55	0.00	0.00	0.00	0.00	0.00	0.04	3.96	0.00
consolidate duplicate cond fragments	ArgoUML	2.34	**2.66**	0.00	0.00	**3.84**	0.00	0.00	0.00	0.00	**2.74**
consolidate duplicate cond fragments	Xerces	**1.40**	1.08	1.84	0.00	**5.33**	**4.49**	0.90	**1.97**	**1.80**	**3.44**
extract method	ApacheAnt	**2.76**	1.83	0.00	0.00	**4.02**	0.00	0.00	0.57	0.00	0.00
extract method	ArgoUML	**12.54**	0.21	0.00	0.00	**9.17**	0.00	0.00	0.00	0.00	0.90
extract method	Xerces	**1.56**	**1.88**	0.43	0.00	**5.94**	0.00	0.00	**4.56**	0.93	**3.47**
extract superclass	ArgoUML	0.00	3.12	4.14	0.00	0.00	0.00	0.00	4.24	0.00	0.00
form template method	ArgoUML	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
inline method	ApacheAnt	2.43	0.84	0.00	0.00	**45.79**	0.00	0.00	1.14	0.00	0.00
inline method	ArgoUML	0.00	1.87	0.00	0.00	0.00	0.00	0.00	85.92	0.00	1.35
inline method	Xerces	**3.29**	1.62	0.00	0.00	**3.23**	0.00	0.00	0.00	0.22	**5.41**
inline temp	ApacheAnt	**8.80**	**2.89**	0.00	0.00	**11.63**	0.00	0.00	0.00	0.00	0.00
inline temp	ArgoUML	1.80	2.02	0.00	0.00	2.71	0.00	0.00	0.00	0.00	0.83
inline temp	Xerces	**2.28**	1.41	1.13	0.00	**5.02**	2.10	0.00	0.00	**2.18**	0.00
introduce assertion	ArgoUML	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.30	0.00	7.40
introduce explaining variable	ApacheAnt	**5.15**	**2.66**	0.00	0.00	**6.88**	4.75	0.00	**4.69**	4.48	0.00
introduce explaining variable	ArgoUML	1.56	0.78	0.00	0.00	**5.06**	0.00	0.00	0.00	0.00	**2.14**
introduce explaining variable	Xerces	1.48	**2.11**	0.00	0.00	**3.73**	**8.17**	0.24	0.98	1.61	2.39
introduce null object	ArgoUML	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
introduce parameter object	Xerces	**14.14**	0.00	0.00	0.00	0.48	0.00	0.00	48.74	0.00	0.00
move field	ApacheAnt	2.49	1.84	0.00	0.00	**43.74**	0.00	0.00	1.82	0.00	0.00
move field	ArgoUML	0.00	1.64	0.00	0.00	0.00	0.00	0.00	22.15	0.00	8.03
move field	Xerces	**1.65**	**0.66**	0.00	**0.53**	**1.27**	0.00	0.00	0.00	**2.04**	0.00
move method	ApacheAnt	1.43	0.49	0.00	0.00	2.92	0.00	0.00	0.00	0.00	0.00
move method	ArgoUML	0.00	1.71	22.35	0.00	0.61	0.00	0.00	0.00	1.13	0.44
move method	Xerces	**2.46**	**0.05**	0.00	0.00	1.10	0.00	0.00	0.00	**0.27**	0.00
pull up field	Xerces	0.00	3.55	19.27	0.00	0.00	0.00	0.00	0.00	0.00	0.00
pull up method	Xerces	**11.95**	0.00	17.86	0.00	0.00	0.00	0.00	0.00	0.00	0.00
push down field	Xerces	**16.43**	0.00	0.00	0.00	**0.21**	0.00	0.00	0.00	0.00	0.00
push down method	Xerces	**26.79**	0.00	16.28	0.00	0.00	0.00	0.00	0.00	0.00	0.00
remove assignment to parameters	ApacheAnt	**3.27**	1.71	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
remove assignment to parameters	ArgoUML	2.36	0.96	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.36
remove assignment to parameters	Xerces	0.92	0.89	**8.79**	0.00	**2.00**	6.47	0.00	0.00	**2.82**	0.00
remove control flag	ApacheAnt	**8.13**	0.67	0.00	0.00	**5.69**	0.00	0.00	0.00	0.00	0.00
remove control flag	ArgoUML	0.40	0.87	**0.08**	0.00	**26.12**	0.00	0.00	0.00	0.54	**9.35**
remove control flag	Xerces	**1.82**	0.85	**3.18**	0.00	**2.85**	0.00	0.53	2.24	**2.29**	0.00
remove parameter	ApacheAnt	**6.54**	**2.17**	0.00	0.00	**5.76**	0.00	0.00	**3.19**	0.00	0.00
remove parameter	ArgoUML	**3.28**	**2.38**	0.00	**0.14**	**3.85**	0.00	0.99	0.00	1.23	0.66
remove parameter	Xerces	1.16	0.97	0.45	0.00	**2.76**	1.38	**0.12**	**2.69**	1.38	1.52
rename method	ApacheAnt	**2.73**	**2.29**	0.00	0.00	**76.36**	0.00	0.00	1.36	0.00	0.00
rename method	ArgoUML	0.00	1.21	0.00	0.00	0.00	0.00	0.00	189.30	2.37	0.54
rename method	Xerces	**14.05**	0.91	0.00	0.00	**1.68**	0.91	**0.10**	0.00	**0.07**	**0.39**
replace data with object	ArgoUML	0.00	4.16	21.65	0.00	0.00	0.00	0.00	43.26	0.00	0.00
replace data with object	Xerces	**2.95**	1.02	0.00	0.00	1.14	0.00	0.00	0.00	**5.07**	0.00
replace exception with test	Xerces	0.74	0.00	0.00	0.00	0.77	0.00	0.00	0.00	0.00	0.00
replace magic number with constant	ApacheAnt	1.14	**3.59**	0.00	0.00	1.01	3.41	0.00	2.12	1.00	0.00
replace magic number with constant	ArgoUML	**4.63**	**17.43**	0.00	0.00	0.00	0.00	0.00	0.00	1.27	0.00
replace magic number with constant	Xerces	**1.31**	**2.08**	0.00	0.00	**2.36**	0.00	**0.12**	0.00	**3.58**	0.45
replace method with method object	ApacheAnt	**16.46**	**4.43**	0.00	0.00	0.00	0.00	0.00	13.70	0.00	0.00
replace method with method object	ArgoUML	0.44	**1.79**	0.00	0.00	**3.90**	0.00	1.11	0.00	0.00	1.14
replace method with method object	Xerces	**3.41**	0.80	0.41	0.00	**1.53**	0.00	0.00	**6.71**	**3.92**	1.05
replace nested cond guard clauses	ApacheAnt	3.09	0.84	0.00	0.00	0.00	0.00	0.00	0.81	0.00	0.00
replace nested cond guard clauses	ArgoUML	0.00	1.48	0.00	0.00	0.00	0.00	0.00	1.40	0.00	**6.52**
replace nested cond guard clauses	Xerces	1.06	0.99	0.00	0.00	**11.34**	0.00	0.46	**3.97**	**2.75**	0.59
separate query from modifier	Xerces	**5.94**	0.00	0.00	0.00	**3.82**	0.00	0.00	0.00	0.00	0.00

by the fact that developers of the three object systems performed a total of 1,753 refactoring operations on classes affected by the Blob code smell (see Table 6.11).

However, the data in Table 6.12 shows that the refactoring operations that actually removed the Blob code smell are mainly two: *move method* and *move field*. Specifically, in Xerces (the only system for which we have a good number of *move method* and *move field* refactoring operations performed on Blob classes), *move method* refactoring removes the Blob code smell in 71% of cases while *move field* refactoring in 30% of cases. By performing a manual analysis of such cases, we discovered as often a set of *move method* refactorings is performed to completely remove a responsibility from the Blob class and, in some cases, *move method* and *move field* refactorings are performed together as *extract class* refactoring (this type of refactoring is not detected by REF-FINDER). For example, the class of the Xerces system named `XSchemaValidator` has been refactored by the developers between releases 1.0.0 and 1.0.4. `XSchemaValidator` was composed of 100 methods and 74 attributes and, as stated in its comment, was an "*experimental implementation of a validator for the W3C schema language*". Developers removed this Blob class from the system by splitting its responsibilities across three new classes extracted from it in release 1.0.4 (i.e., `Schema`, `SchemaImporter`, and `SchemaParser`). This was done by (i) partially rewriting the code present in class `XSchemaValidator`, and (ii) by performing 52 *move field* and 31 *move method* operations from `XSchemaValidator` to the three new extracted classes.

Thus, while a Blob class generally represents a catalyst of several refactoring operations due to its size (i.e., high LOCs), *move method* and *move field* refactorings (or in combination as *extract class*) seem to be the only refactoring operations effective in removing this design problem from the system.

Classes affected by the Class Data Should Be Private (CDSBP) code smell also attracted several refactoring operations. However, it is worth noting that this is mainly due to the fact that this is the most diffused code smell we found (see Table 6.9). In fact, as shown in Table 6.12, no refactoring operations removed this code smell. The refactoring operation having this goal is the *encapsulate field*. However, we only found one instance of this refactoring in the ArgoUML system. What in-

Table 6.11: Number of refactorings performed on each type of code smell.

Refactoring	System	Blob	CDSBP	Complex Class	Lazy Class	Long Method	LPL	Refused Bequest	Spaghetti Code	Speculative Generality	Feature Envy
add parameter	ApacheAnt	20	18	0	0	33	0	0	24	0	0
add parameter	ArgoUML	15	23	0	1	13	0	1	0	4	19
add parameter	Xerces	131	63	3	0	201	5	3	18	71	5
consolidate cond expression	ApacheAnt	5	2	0	0	3	0	0	3	0	0
consolidate cond expression	ArgoUML	2	1	0	0	2	0	0	0	0	6
consolidate cond expression	Xerces	46	15	0	0	44	3	2	1	10	0
consolidate duplicate cond fragments	ApacheAnt	6	9	0	0	0	0	0	0	1	0
consolidate duplicate cond fragments	ArgoUML	3	6	0	0	3	0	0	0	0	9
consolidate duplicate cond fragments	Xerces	125	43	10	0	193	6	9	12	33	15
extract method	ApacheAnt	7	10	0	0	9	0	0	2	0	0
extract method	ArgoUML	16	1	0	0	10	0	0	0	0	6
extract method	Xerces	53	24	1	0	84	0	0	11	10	7
extract superclass	ArgoUML	0	1	0	0	0	0	0	0	0	0
form template method	ArgoUML	0	0	0	0	0	0	0	0	0	0
inline method	ApacheAnt	2	1	0	0	16	0	0	6	0	0
inline method	ArgoUML	0	1	0	0	0	0	0	0	0	1
inline method	Xerces	31	8	0	0	31	0	0	0	1	6
inline temp	ApacheAnt	15	13	0	0	12	0	0	0	0	0
inline temp	ArgoUML	2	5	0	0	2	0	0	0	0	3
inline temp	Xerces	37	15	2	0	47	1	0	0	12	0
introduce assertion	ArgoUML	0	0	0	0	0	0	0	0	0	3
introduce explaining variable	ApacheAnt	17	19	0	0	38	1	0	24	1	0
introduce explaining variable	ArgoUML	2	2	0	0	4	0	0	0	0	8
introduce explaining variable	Xerces	45	29	0	0	63	4	1	2	14	4
introduce null object	ArgoUML	0	0	0	0	0	0	0	0	0	0
introduce parameter object	Xerces	8	0	0	0	4	0	0	3	0	0
move field	ApacheAnt	5	5	0	0	43	0	0	20	0	0
move field	ArgoUML	0	13	0	0	0	0	0	0	0	0
move field	Xerces	53	4	0	0	42	0	0	0	5	0
move method	ApacheAnt	1	1	0	0	2	0	0	0	0	0
move method	ArgoUML	0	15	3	0	4	0	0	0	2	4
move method	Xerces	71	3	0	0	62	0	0	0	3	0
pull up field	Xerces	0	0	0	0	0	0	0	0	0	0
pull up method	Xerces	5	0	0	0	0	0	0	0	0	0
push down field	Xerces	9	0	0	0	7	0	0	0	0	0
push down method	Xerces	0	0	0	0	0	0	0	0	0	0
remove assignment to parameters	ApacheAnt	5	7	0	0	0	0	0	0	0	0
remove assignment to parameters	ArgoUML	1	1	0	0	0	0	0	0	0	2
remove assignment to parameters	Xerces	12	6	4	0	18	1	0	0	6	0
remove control flag	ApacheAnt	5	2	0	0	4	0	0	0	0	0
remove control flag	ArgoUML	1	3	1	0	21	0	0	0	0	44
remove control flag	Xerces	26	4	0	0	42	0	0	0	5	0
remove parameter	ApacheAnt	20	19	0	0	29	0	0	16	0	0
remove parameter	ArgoUML	16	26	0	1	12	0	1	0	5	11
remove parameter	Xerces	91	42	2	0	140	2	2	15	22	7
rename method	ApacheAnt	13	14	0	0	112	0	0	44	0	0
rename method	ArgoUML	0	8	0	0	0	0	0	0	5	5
rename method	Xerces	563	62	0	0	339	6	2	0	4	9
replace data with object	ArgoUML	0	1	0	0	0	0	0	0	0	0
replace data with object	Xerces	11	5	0	0	9	0	0	0	5	0
replace exception with test	Xerces	1	0	0	0	1	0	0	0	0	0
replace magic number with constant	ApacheAnt	17	62	0	0	16	1	0	8	1	0
replace magic number with constant	ArgoUML	9	46	0	0	0	0	0	0	2	0
replace magic number with constant	Xerces	104	101	0	0	144	0	2	0	64	2
replace method with method object	ApacheAnt	17	17	0	0	0	0	0	0	0	0
replace method with method object	ArgoUML	2	16	0	0	10	0	1	0	0	15
replace method with method object	Xerces	59	17	1	0	55	0	0	9	18	2
replace nested cond guard clauses	ApacheAnt	1	1	0	0	0	0	0	0	0	0
replace nested cond guard clauses	ArgoUML	0	1	0	0	0	0	0	0	0	6
replace nested cond guard clauses	Xerces	37	16	0	0	79	0	1	10	17	1
separate query from modifier	Xerces	10	0	0	0	9	0	0	0	0	0

Table 6.12: Perc. of smells removed by each refactoring. In **bold** values for which a refactoring has been applied at least 10 times on a smell.

Refactoring	System	Blob	CDSBP	Complex Class	Lazy Class	Long Method	LPL	Refused Bequest	Spaghetti Code	Speculative Generality	Feature Envy
add parameter	ApacheAnt	**0%**	**0%**	0%	0%	**0%**	0%	0%	**100%**	0%	0%
add parameter	ArgoUML	**0%**	**0%**	0%	0%	**15%**	0%	100%	0%	50%	**16%**
add parameter	Xerces	**4%**	**0%**	0%	0%	**6%**	0%	0%	**6%**	**7%**	100%
consolidate cond expression	ApacheAnt	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%
consolidate cond expression	ArgoUML	0%	0%	0%	0%	0%	0%	0%	0%	0%	17%
consolidate cond expression	Xerces	**2%**	**0%**	0%	0%	**2%**	0%	0%	0%	**0%**	0%
consolidate duplicate cond fragments	ApacheAnt	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
consolidate duplicate cond fragments	ArgoUML	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
consolidate duplicate cond fragments	Xerces	**2%**	**0%**	**0%**	0%	**1%**	0%	0%	**25%**	**12%**	**93%**
extract method	ApacheAnt	0%	**0%**	0%	0%	22%	0%	0%	100%	0%	0%
extract method	ArgoUML	**0%**	0%	0%	0%	**40%**	0%	0%	0%	0%	50%
extract method	Xerces	**0%**	**0%**	0%	0%	**11%**	0%	0%	**0%**	**0%**	100%
extract superclass	ArgoUML	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
form template method	ArgoUML	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
inline method	ApacheAnt	0%	0%	0%	0%	**0%**	0%	0%	100%	0%	0%
inline method	ArgoUML	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
inline method	Xerces	**3%**	0%	0%	0%	**3%**	0%	0%	0%	0%	100%
inline temp	ApacheAnt	**0%**	**0%**	0%	0%	**0%**	0%	0%	0%	0%	0%
inline temp	ArgoUML	0%	0%	0%	0%	0%	0%	0%	0%	0%	33%
inline temp	Xerces	**0%**	**0%**	0%	0%	**0%**	0%	0%	0%	**0%**	0%
introduce assertion	ArgoUML	0%	0%	0%	0%	0%	0%	0%	0%	0%	67%
introduce explaining variable	ApacheAnt	**0%**	**0%**	0%	0%	**0%**	0%	0%	**100%**	0%	0%
introduce explaining variable	ArgoUML	0%	0%	0%	0%	0%	0%	0%	0%	0%	13%
introduce explaining variable	Xerces	**0%**	**0%**	0%	0%	**3%**	0%	0%	0%	**0%**	100%
introduce null object	ArgoUML	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
introduce parameter object	Xerces	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
move field	ApacheAnt	0%	0%	0%	0%	**0%**	0%	0%	**100%**	0%	0%
move field	ArgoUML	0%	**0%**	0%	0%	0%	0%	0%	0%	0%	0%
move field	Xerces	**30%**	0%	0%	0%	**0%**	0%	0%	0%	0%	0%
move method	ApacheAnt	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
move method	ArgoUML	0%	**0%**	0%	0%	0%	0%	0%	0%	100%	0%
move method	Xerces	**73%**	0%	0%	0%	**0%**	0%	0%	0%	0%	0%
pull up field	Xerces	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
pull up method	Xerces	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%
push down field	Xerces	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
push down method	Xerces	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
remove assignment to parameters	ApacheAnt	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
remove assignment to parameters	ArgoUML	0%	0%	0%	0%	0%	0%	0%	0%	0%	50%
remove assignment to parameters	Xerces	**0%**	0%	0%	0%	**0%**	0%	0%	0%	0%	0%
remove control flag	ApacheAnt	0%	0%	0%	0%	25%	0%	0%	0%	0%	0%
remove control flag	ArgoUML	0%	0%	0%	0%	**81%**	0%	0%	0%	0%	**9%**
remove control flag	Xerces	**4%**	0%	0%	0%	**14%**	0%	0%	0%	0%	0%
remove parameter	ApacheAnt	**0%**	**0%**	0%	0%	**0%**	0%	0%	**100%**	0%	0%
remove parameter	ArgoUML	**0%**	**0%**	0%	0%	**7%**	0%	100%	0%	60%	**27%**
remove parameter	Xerces	**5%**	**0%**	0%	0%	**9%**	0%	0%	**0%**	**9%**	100%
rename method	ApacheAnt	**0%**	**0%**	0%	0%	**0%**	0%	0%	100%	0%	0%
rename method	ArgoUML	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%
rename method	Xerces	**9%**	**0%**	0%	0%	**57%**	0%	0%	0%	0%	100%
replace data with object	ArgoUML	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
replace data with object	Xerces	**7%**	0%	0%	0%	11%	0%	0%	0%	0%	0%
replace exception with test	Xerces	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
replace magic number with constant	ApacheAnt	**0%**	**0%**	0%	0%	**0%**	0%	0%	100%	0%	0%
replace magic number with constant	ArgoUML	0%	**0%**	0%	0%	0%	0%	0%	0%	0%	0%
replace magic number with constant	Xerces	**0%**	**0%**	0%	0%	**4%**	0%	0%	0%	**19%**	100%
replace method with method object	ApacheAnt	**0%**	**0%**	0%	0%	0%	0%	0%	0%	0%	0%
replace method with method object	ArgoUML	0%	**0%**	0%	0%	**30%**	0%	100%	0%	0%	**13%**
replace method with method object	Xerces	**5%**	**0%**	0%	0%	**4%**	0%	0%	0%	**0%**	100%
replace nested cond guard clauses	ApacheAnt	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
replace nested cond guard clauses	ArgoUML	0%	0%	0%	0%	0%	0%	0%	0%	0%	17%
replace nested cond guard clauses	Xerces	**0%**	**0%**	0%	0%	**0%**	0%	0%	**0%**	**0%**	100%
separate query from modifier	Xerces	**0%**	0%	0%	0%	22%	0%	0%	0%	0%	0%

stead stands out from the analysis of the ORs reported in Table 6.10, is that classes affected by CDSBP have a much higher chance of being involved in *replace magic number with constant* refactoring operations (this chance is up to 17.43 higher). By manually analyzing those cases, we did not find a clear explanation for this phenomenon. However, two possible explanations are plausible from our point of view. The first is that developers are more prone to add new class fields (and thus to apply *replace magic number with constant* refactoring) in classes already containing fields (like those affected by the CDSBP code smell). The second is that the introduction of this code smell is favored by the application of the *replace magic number with constant* refactoring. Indeed, such refactoring implies the introduction of a new field within the class and it is possible that the added field is publicly exposed, introducing a CDSBP.

Particularly interesting are the results achieved for Complex and Lazy Classes. Both are poorly refactored by developers. On the one side, Lazy Classes are very simple classes, thus they should not create too much trouble during maintenance activities, and consequently developers are not particularly motivated to refactor them. For example, the interface `LayoutedObject` from ArgoUML reported in Listing 6.1 has never been refactored by ArgoUML developers until the last release considered in our study (0.34). Hence, this is an expected result.

```
1 package org.argouml.uml.diagram.layout;
  // This is the most common form of an layouted
3 // object.
  public interface LayoutedObject {
5 }
```

Listing 6.1: Example of a Lazy Class never refactored by developers in ArgoUML

On the other side, the reason behind the very few refactorings performed on Complex Classes is likely their complexity. In total, we observed just 27 refactoring operations on the 115 complex classes involved in our study (to be compared, as example, to the 1,753 performed on the 609 Blob classes). For example, the Complex Class `RegularExpression` from the Xerces system has never been

refactored by the developers. By looking inside its source code we found that `RegularExpression` is a large class composed of 3,155 LOCs, and the 32 methods contained in it are very complex. To get an idea, these methods contain in total 126 `switch case` statements and 536 `if else` statements. Thus, refactoring this class would be very challenging for developers.

Conversely, classes containing Long Methods are widely refactored, for a total of 2,012 total refactorings. Firstly, it is interesting to note that 35% of classes affected by Long Methods are also Blobs and, as these latter, they also catalyze the refactoring attention of developers. In particular, classes affected by this code smell have:

- from 2.27 to 3.62 times more chances of being involved in an *add parameter* refactoring;
- from 4.02 to 9.17 times more chances of being involved in an *extract method* refactoring;
- form 3.23 to 45.79 times more chances of being involved in an *inline method* refactoring (no data for ArgoUML);
- from 3.73 to 6.88 times more chances of being involved in an *introducing explaining variable* refactoring;
- from 2.85 to 26.12 times more chances of being involved in a *remove control flag* refactoring;
- from 2.76 to 5.76 times more chances of being involved in a *remove parameter* refactoring;
- from 1.68 to 76.36 times more chances of being involved in a *rename method* refactoring (no data for ArgoUML).

However, as shown in Table 6.12, only some of these refactorings are applied by developers with the aim of removing the Long Method. The refactoring more often removing a Long Method is the *remove control flag* that helps in removing

the code smell by reducing the method length. As expected, the other refactoring often removing the Long Method is the *extract method*, representing the most natural solution to this code smell. This refactoring has been applied by Xerces developers between release 2.7.1 and release 2.8.0 on the Long Method `DOMSerializerImpl.writeToString(Node wnode)` to extract from it three new methods (i.e., `_getXmlVersion(Node node)`, `_getInputEncoding(Node node)`, `_getXmlEncoding(Node node)`), each one implementing a specific responsibility.

Table 6.13: Code Smell model: Summary of the achieved results

Code Smell	Refactoring operations related to the metric		Overlap
	Expected	Found	
Blob	extract class; move method; move field	extract class (as combination of move method and move field); move method; move field	100%
CDSBP	encapsulate field	replace magic number with constant	0%
Complex Class	extract method; consolidate conditional expression; move method; extract class	-	0%
Lazy Class	inline class	-	0%
Long Method	extract method; remove control flag; consolidate conditional expression	extract method; remove control flag; consolidate conditional expression; add parameter; remove parameter; inline method; introducing explaining variable; rename method	38%
LPL	introduce parameter object	-	0%
Refused Bequest	push down method; push down field; replace inheritance with delegation	-	0%
Spaghetti Code	add parameter	add parameter; remove parameter	50%
Speculative Generality	collapse hierarchy	-	0%
Feature Envy	move method; extract method; consolidate duplicate conditional fragments	consolidate duplicate conditional fragments	33%

It can also be noted that the high number of *extract method* refactorings partially explains the high number of *rename method* refactorings performed on long methods. Indeed, the method undergoing an *extract method* refactoring is generally also renamed to reflect its new purpose. As for the *add parameter* refactoring, it sometimes helps to remove a Long Method. This is due to the fact that computations

previously performed inside the method to obtain a result r are now required to the classes invoking the long method through the passing of r as parameter.

As for the other refactorings previously mentioned (i.e., *inline method*, *introducing explaining variable*, *remove parameter*) they are massively performed on classes affected by Long Method mainly due to the long size of the involved code component.

The Long Parameter List (LPL) code smell is rarely refactored by developers (just 30 refactorings in total) as well as the Refused Bequest code smell (25 refactorings). Classes affected by the Spaghetti Code code smell have a higher chance of being involved in an *add parameter* refactoring. This is a very expected result. In fact, these classes are generally composed by methods with few (or no) parameters. Note that, as shown in Table 6.12, this refactoring is able to remove the code smell in 100% of cases on ApacheAnt. However, a deeper analysis, reported in Table 6.12, reveals that also the *remove parameter* refactoring removes the Spaghetti Code code smell in 100% of cases on ApacheAnt. Our manual analysis revealed that the 16 *remove parameter* performed on Spaghetti Code in ApacheAnt were always executed together with an *add parameter* refactoring. In particular, the parameter was generally moved from methods having more than one parameter to methods having no parameters inside the same class.

For the Speculative Generality code smell, we did not observe any particular result, while it is interesting to note that in 93% of cases a *consolidate duplicate conditional fragments* refactoring operation is able to remove a Feature Envy code smell on Xerces (the only system on which we have data for this refactoring). This refactoring removes a fragment of code that is present in more than one branch of a conditional expression. This means that often, a high coupling between one method and the "envied class" (i.e., the class causing the Feature Envy in which the method should moved) is not really needed, but just emphasized by duplicated code.

In summary, *5,425 of the analyzed 12,922 refactoring operations are performed on code smells (42%)*. However, of these 5,425 *only 933 actually removed the code smell from the affected class (7% of total operations)* and 895 are attributable to only four

code smells (i.e., Blob, Long Method, Spaghetti Code, and Feature Envy). Table 6.13 summarizes our findings for the studied code smells, highlighting for each of them (i) the refactoring operations for which we expected a correlation with the presence of code smells, (ii) the refactoring operations that we identified as applied on the code smell and able to often remove it, and (iii) the percentage overlap between the two previous explained sets. Looking at Table 6.13 we conclude that:

- *Only some of the analyzed code smells, such as Blob, Long Method, Spaghetti Code, and Feature Envy, actually increase the chances of the affected classes of being refactored*.

- *The effectiveness of refactoring operations in removing code smells is generally low. In the analyzed project releases, only 7% of the smells are removed through refactoring operations.*

6.4 Threats to Validity

This section discusses the threats that could affect the validity of our study. Threats to *construct validity* concern the relationship between theory and observation. The most important threat to construct validity to be discussed is how we assess source code quality in this chapter. Specifically, we have chosen to use source code metrics, namely LOC, Chidamber & Kemerer metrics, conceptual cohesion and coupling. Clearly, there may be other metrics that may capture software quality, for example metrics computed by means of dynamic analysis. Nevertheless, as explained in Section 6.2.2, we have chosen a mix of metrics capturing source code size, structural and lexical characteristics. Another threat to validity concerns the identification of code smells. As explained in Section 6.2.2, we used a constraint-based approach to perform a preliminary detection of code smells (using low threshold values to avoid reducing the recall) followed by a manual analysis performed by two independent evaluators (with the aim of reducing imprecision and subjectiveness). Despite such process, we cannot exclude that some code smells were missed by our analysis or that false positives were considered. Finally, sim-

ilar issues apply to the investigated refactorings, selected through a manual validation over an initial set detected by REF-FINDER. As pointed out by its authors [172], REF-FINDER has a very good recall (95%) while the precision is a bit lower (79%). However, in this study we back-up possible imprecisions by complementing REF-FINDER by manual validation.

Threats to *conclusion validity* concern the relationship between treatment and outcome. We use logistic regression models to identify correlations between metric values, and the presence of code smells with refactoring actions. Other than highlighting cases of significant correlations, we report and discuss OR values.

Threats to *internal validity* concern factors that could influence our observations. In particular, the fact that code smells disappear, may or may not be related to refactoring activities occurred between the observed releases. In other words, other changes could have produced such effects. However, although the performed analyses and the obtained results allow us to claim correlation and not causation, we corroborate our quantitative results by means of some qualitative analysis, aimed at illustrating examples in which specific kinds of refactorings helped to remove some code smells.

Threats to *external validity* concern the generalization of our findings. The study is limited to three Java projects, because we preferred to observe fewer projects over a long period of evolution history, rather than many projects for a short period. This better allowed us to observe refactorings, that often happen during specific periods of a project lifetime [8]. We considered open source systems for our analysis, since the source code of commercial ones are not available. However, we provided data and tools used for the investigation in order to allow a replication on different (both open source and commercial) systems. Last, but not least, as mentioned in Section 6.2, this choice to analyze few systems was also due to the need for manually validating refactorings and smells, rather than just relying on tool output. In any case, further studies are therefore needed to confirm (or refute) our results. Also, the findings obtained for the investigated code smells may or may not apply to other kinds of code smells, for example those—such as *Divergent Change* or *Parallel Inheritance*—that can be detected using change history metrics

[52].

6.5 Conclusion

This chapter reported an empirical study aimed at investigating the characteristics of code components increasing their changes of being subject to refactoring operations. In particular, we verified whether refactoring activities occur on classes for which certain indicators—such as quality metrics or the presence of smells as detected by tools—suggest there might be need for refactorings. The study has been conducted on 63 releases of three open source projects, and required the manual analysis of 15,008 refactoring operations and 5,478 smells.

Our results highlighted that, with very few exceptions, quality metrics do not show a clear relationship with refactoring. One possible interpretation of such a finding can be found in our survey performed with developers about their perception about some code smells [50] (presented in Chapter 5). Indeed, on the one hand developers found that only particularly serious smells (in terms of metrics) are worthwhile of being refactored. On the other hand, they also pointed out that in some cases metrics may not be *per se* indicators of smells: for example, some classes—*e.g.,* implementing parsers or complex algorithms—might intrinsically exhibit anomalous metric profiles, without necessarily being considered as refactoring opportunities.

Almost 40% of the analyzed refactorings has been performed on classes affected by smells. However, just 7% of them actually removed the smell. In other words, it is possible that the refactoring only *mitigated* the problem, without however necessarily removing completely the smell.

This work is mainly exploratory in nature, as it is aimed at empirically investigating a phenomenon—which characteristics of classes promote refactoring operations—from a **quantitative** point-of-view. Nevertheless, there are different possible uses one can make of the results of this chapter. When *building* recommendation tools aimed at highlighting refactoring opportunities to developers it must be taken into account that, at least among the code characteristics considered

in this chapter—*i.e.*, code metrics, presence of smells—there is no silver bullet able to indicate which code artifacts are in need of refactoring. Future work in this area should aim at learning something from the past refactorings made by developers, in order to suggest refactoring recommendations more suitable for them.

Also, when *evaluating* refactoring recommendation tools the developer's point-of-view cannot be ignored. Often such tools are just evaluated by verifying if the refactorings they recommend are able to improve some quality metric values and/or to remove smells. However, our study indicates that the developer's point-of-view of classes in need of refactoring does not always match with these "quality indicators".

Part 3

Alternative Sources of Information for Code Smell Detection

Chapter 7

Mining Version Histories for Detecting Code Smells

7.1 Introduction

Code smells have been defined by Fowler [8] as symptoms of poor design and implementation choices. In some cases, such symptoms may originate from activities performed by developers while in a hurry, *e.g.,* implementing urgent patches or simply making suboptimal choices. In other cases, smells come from some recurring, poor design solutions, also known as anti-patterns [61]. For example a *Blob* is a large and complex class that centralizes the behavior of a portion of a system and only uses other classes as data holders. *Blob* classes can rapidly grow out of control, making it harder and harder for developers to understand them, to fix bugs, and to add new features.

Previous studies have found that smells hinder comprehension [18], and possibly increase change- and fault-proneness [16, 17]. In summary, smells need to be carefully detected and monitored and, whenever necessary, refactoring actions should be planned and performed to deal with them.

There exist a number of approaches for detecting smells in source code to alert developers of their presence [21, 20, 25]. These approaches rely on structural information extracted from source code, for example, by means of constraints defined

on some source code metrics. For instance, according to some existing approaches, such as DECOR [20], *LongMethod* or *LargeClass* smells are based on the size of the source code component in terms of LOC, whereas other smells like *ComplexClass* are based on the McCabe cyclomatic complexity [186].

Although existing approaches exhibit good detection accuracy, they still might not be adequate for detecting many of the smells described by Fowler [8]. In particular, there are some smells that, rather than being characterized by source code metrics or other information extracted from source code snapshots, are *intrinsically characterized by how source code changes over time*. For example, a *Parallel Inheritance* means that two or more class hierarchies evolve by adding code to both classes at the same time. Also, there are smells that are traditionally detected using structural information, where historical information can aid in capturing complementary, additionally useful properties. For example, a *Feature Envy* may manifest itself when a method of a class tends to change more frequently with methods of other classes rather than with those of the same class.

Based on such considerations, we propose HIST (**H**istorical **I**nformation for **S**mell de**T**ection), an approach to detect smells based on change history information mined from versioning systems, and, specifically, by analyzing co-changes occurring between source code artifacts. HIST is aimed at detecting five smells from Fowler [8] and Brown [61] catalogues. Three of them—*Divergent Change*, *Shotgun Surgery*, and *Parallel Inheritance*—are symptoms that can be intrinsically observed from the project's history even if a single project snapshot detection approach has been proposed for the detection of *Divergent Change* and *Shotgun Surgery* [79]. For the remaining two—*Blob* and *Feature Envy*—there exist several single project snapshot detection approaches [20, 25]. However, as explained for the *Feature Envy*, those smells can also be characterized and possibly detected using source code change history.

In the past, historical information has been used in the context of smell analysis for the purpose of assessing to what extent smells remained in the system for a substantial amount of time [15, 12]. Also, Gîrba *et al.* [187] exploited formal concept analysis for detecting co-change patterns, that can be used to detect some

smells. However, to the best of our knowledge, the use of historical information for smell detection remains a premiere of this thesis.

We have evaluated HIST on twenty Java projects, aimed at evaluating its detection accuracy in terms of precision and recall against a manually-produced oracle. Furthermore, wherever possible, we compared HIST with results produced by approaches that detect smells by analyzing a single project snapshot, such as JDeodorant [25, 67] (for the *Feature Envy* smell) and our re-implementations of the *DECOR's* [20] detection rules (for the *Blob* smell) and of the approach by Rao *et al.* [79] (for *Divergent Change* and *Shotgun Surgery*). The results of our study indicate that HIST's precision is between 72% and 86%, and its recall is between 58% and 100%. When comparing HIST to alternative approaches, we observe that HIST tends to provide better detection accuracy, especially in terms of recall, since it is able to identify smells that other approaches omit. Also, for some smells, we observe a strong complementarity of the approaches based on a single snapshot analysis with respect to HIST, suggesting that even better performances can be achieved by combining these two complementary sources of information.

7.2 Detecting Code Smells Using Change History Information

The key idea behind HIST is to identify classes affected by smells via change history information derived from version control systems. Fig. 7.1 overviews the main steps behind the proposed approach. Firstly, HIST extracts information needed to detect smells from the versioning system through a component called *Change history extractor*. This information—together with a specific detection algorithm for a particular smell—is then provided as an input to the *Code smell detector* for computing the list of code components (*i.e.*, methods/classes) affected by the smells characterized in the specific detection algorithm.

The *Code smell detector* uses different detection heuristics for identifying target smells. We have instantiated HIST for detecting the five smells summarized in the

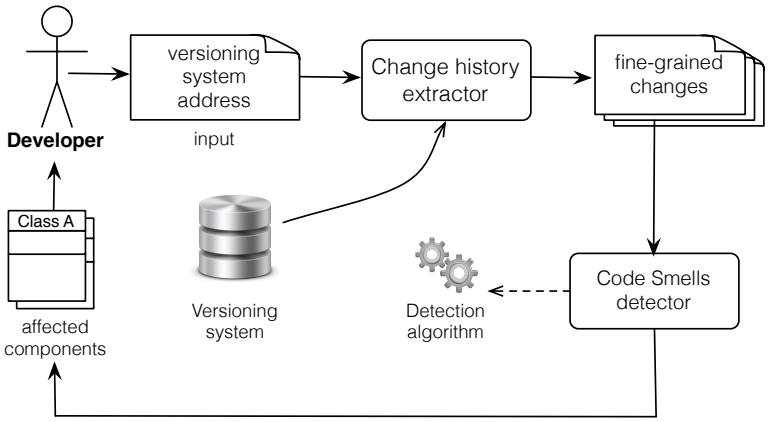


Figure 7.1: HIST: The proposed code smell detection process.

following:

- *Divergent Change*: this smell occurs when a class is changed in different ways for different reasons. The example reported by Fowler in his book on refactoring [8] helps understanding this smell: *If you look at a class and say, "Well, I will have to change these three methods every time I get a new database; I have to change these four methods every time there is a new financial instrument", you likely have a situation in which two classes are better than one* [8]. Thus, this type of smell clearly triggers *Extract Class* refactoring opportunities[1]. Indeed, the goal of *Extract Class* refactoring is to split a class implementing different responsibilities into separated classes, each one grouping together methods and attributes related to a specific responsibility. The aim is to (i) obtain smaller classes that are easier to comprehend and thus to maintain and (ii) better isolate the change.

- *Shotgun Surgery*: a class is affected by this smell when a change to this class (*i.e.,* to one of its fields/methods) triggers many little changes to several other classes [8]. The presence of a Shotgun Surgery smell can be removed through

[1]Further details about refactoring operations existing in the literature can be found in the refactoring catalog available at:
`http://refactoring.com/catalog/`

a *Move Method/Field* refactoring. In other words, the method/field causing the smell is moved towards the class in which its changes trigger more modifications.

- *Parallel Inheritance*: this smell occurs when *"every time you make a subclass of one class, you also have to make a subclass of another"* [8]. This could be symptom of design problems in the class hierarchy that can be solved by redistributing responsibilities among the classes through different refactoring operations, *e.g., Extract Subclass*.

- *Blob*: a class implementing several responsibilities, having a large number of attributes, operations, and dependencies with data classes [61]. The obvious way to remove this smell is to use *Extract Class* refactoring.

- *Feature Envy*: as defined by Fowler [8], this smell occurs when *"a method is more interested in another class than the one it is actually in"*. For instance, there can be a method that frequently invokes accessor methods of another class to use its data. This smell can be removed via *Move Method* refactoring operations.

Our choice of instantiating the proposed approach on these smells is not random, but driven by the need to have a benchmark including smells that can be naturally identified using change history information and smells that do not necessarily require this type of information. The first three smells, namely *Divergent Change*, *Shotgun Surgery*, and *Parallel Inheritance*, are by definition historical smells, that is, their definition inherently suggests that they can be detected using revision history. Instead, the last two smells (*Blob* and *Feature Envy*) can be detected relying solely on structural information, and several approaches based on static source code analysis of a single system's snapshot have been proposed for detecting those smells [20, 25].

The following subsections detail how HIST extracts change history information from versioning systems and then uses it for detecting the above smells.

7.2.1 Change History Extraction

The first operation performed by the *Change history extractor* is to mine the versioning system log, reporting the entire change history of the system under analysis. This can be done for a range of versioning systems, such as `SVN`, `CVS`, or `Git`. However, the logs extracted through this operation report code changes at file level of granularity. Such a granularity level is not sufficient to detect most of the smells defined in the literature. In fact, many of them describe method-level behavior (see, for instance, *Feature Envy* or *Divergent Change*)[2]. In order to extract fine-grained changes, the *Change history extractor* includes a code analyzer component that is developed in the context of the MARKOS European project[3]. We use this component to capture changes at method level granularity. In particular, for each pair of subsequent source code snapshots extracted from the versioning system, the code analyzer (i) checks out the two snapshots in two separate folders and (ii) compares the source code of these two snapshots, producing the set of changes performed between them. The set of changes includes: (i) added/removed/moved/renamed classes, (ii) added/removed class attributes, (iii) added/removed/moved/renamed methods, (iv) changes applied to all the method signatures (i.e., visibility change, return type change, parameter added, parameter removed, parameter type change, method rename), and (v) changes applied to all the method bodies.

The code analyzer parses source code by relying on the `srcML` toolkit [188]. To distinguish cases where a method/class was removed and a new one added from cases when a method/class was moved (and possibly its source code changed), the MARKOS code analyzer uses heuristics that map methods/classes with different names if their source code is similar based on a metric fingerprint similar to the one used in metric-based clone detection [189]. For example, each method is associated with a twelve digits fingerprint containing the following information: LOCs,

[2]Note that some versioning systems allow to obtain line diffs of the changes performed in a commit. However, the mapping between the changed lines and the impacted code components (*e.g.*, which methods are impacted by the change) is not provided.

[3]`www.markosproject.eu` verified on September 2014

number of statements, number of *if* statements, number of *while* statements, number of *case* statements, number of *return* statements, number of specifiers, number of parameters, number of thrown exceptions, number of declared local variables, number of method invocations, and number of used class attributes (*i.e.,* instance variables). The accuracy of such heuristics has been evaluated at two different levels of granularity:

- *Method level*, by manually checking 100 methods reported as moved by the MARKOS code analyzer. Results showed that 89 of them were actually moved methods.

- *Class level*, by manually checking 100 classes reported as moved by the code analyzer. Results showed that 98 of them were actually moved classes.

Typical cases of false positives were those in which a method/class was removed from a class/package and a very similar one—in terms of fingerprint—was added to another class/package.

7.2.2 Code Smells Detection

The set of fine-grained changes computed by the *Change history extractor* is provided as an input to the *Code Smell detector*, that identifies the list of code components (if any) affected by specific smells. While the exploited underlying information is the same for all target smells (*i.e.,* the change history information), HIST uses custom detection heuristics for each smell. Note that, since HIST relies on the analysis of change history information, it is possible that a class/method that behaved as affected by a smell in the past does not exist in the current version of the system, *e.g.,* because it has been refactored by the developers. Thus, once HIST identifies a component that is affected by a smell, HIST checks the presence of this component in the current version of the system under analysis before presenting the results to the user. If the component does not exist anymore, HIST removes it from the list of components affected by smells.

In the following we describe the heuristics we devised for detecting the different kinds of smells described above, while the process for calibrating the heuristic parameters is described in Section 7.3.1.

Divergent Change Detection

Given the definition of this smell provided by Fowler [8], our conjecture is that *classes affected by Divergent Change present different sets of methods each one containing methods changing together but independently from methods in the other sets*. The *Code Smell detector* mines association rules [190] for detecting subsets of methods in the same class that often change together. Association rule discovery is an unsupervised learning technique used for local pattern detection highlighting attribute value conditions that occur together in a given dataset [190]. In HIST, the dataset is composed of a sequence of change sets—*e.g.*, methods—that have been committed (changed) together in a version control repository [191]. An association rule, $M_{left} \Rightarrow M_{right}$, between two disjoint method sets implies that, if a change occurs in each $m_i \in M_{left}$, then another change should happen in each $m_j \in M_{right}$ within the same change set. The strength of an association rule is determined by its support and confidence [190]:

$$\textit{Support} = \frac{|M_{left} \cup M_{right}|}{T} \tag{7.1}$$

$$\textit{Confidence} = \frac{|M_{left} \cup M_{right}|}{|M_{left}|} \tag{7.2}$$

where T is the total number of change sets extracted from the repository. In this chapter, we perform association rule mining using a well-known algorithm, namely `Apriori` [190]. Note that, minimum *Support* and *Confidence* to consider an association rule as valid can be set in the `Apriori` algorithm. Once HIST detects these change rules between methods of the same class, it identifies classes affected by *Divergent Change* as those containing at least two sets of methods with the following characteristics:

1. The cardinality of the set is at least γ;

2. All methods in the set change together, as detected by the association rules; and

3. Each method in the set does not change with methods in other sets as detected by the association rules.

Shotgun Surgery Detection

In order to define a detection strategy for this smell, we exploited the following conjecture: *a class affected by Shotgun Surgery contains at least one method changing together with several other methods contained in other classes*. Also in this case, the *Code Smell detector* uses association rules for detecting methods—in this case methods from different classes—often changing together. Hence, a class is identified as affected by a *Shotgun Surgery* smell if it contains at least one method that changes with methods present in more than δ different classes.

Parallel Inheritance Detection

Two classes are affected by *Parallel Inheritance* smell if *"every time you make a subclass of one class, you also have to make a subclass of the other"* [8]. Thus, the *Code Smell detector* identifies pairs of classes for which the addition of a subclass for one class implies the addition of a subclass for the other class using generated association rules. These pairs of classes are candidates to be affected by the *Parallel Inheritance* smell.

Blob Detection

A *Blob* is a class that centralizes most of the system's behavior and has dependencies towards data classes [61]. Thus, our conjecture is that *despite the kind of change developers have to perform in a software system, if a* Blob *class is present, it is very likely that something will need to be changed in it*. Given this conjecture, Blobs are identified as classes modified (in any way) in more than $\alpha\%$ of commits involving at

least another class. This last condition is used to better reflect the nature of *Blob* classes that are expected to change despite the type of change being applied, *i.e.*, the set of modified classes.

Feature Envy Detection

Our goal here is to identify methods placed in the wrong class or, in other words, methods having an envied class which they should be moved into. Thus, our conjecture is that *a method affected by feature envy changes more often with the envied class than with the class it is actually in*. Given this conjecture, HIST identifies methods affected by this smell as those involved in commits with methods of another class of the system $\beta\%$ more than in commits with methods of their class.

7.3 The Accuracy of HIST

The *goal* of the study is to evaluate HIST, with the *purpose* of analyzing its effectiveness in detecting smells in software systems. The *quality focus* is on the detection accuracy and completeness as compared to the approaches based on the analysis of a single project snapshot, while the *perspective* is of researchers, who want to evaluate the effectiveness of historical information in identifying smells for building better recommenders for developers.

7.3.1 Study Design

This section provides details about the design and planning of the study aimed at assessing HIST's effectiveness and comparing it with alternative approaches.

Context Selection

The *context* of the study consists of twenty software projects. Table 7.1 reports the characteristics of the analyzed systems, namely the software history that we investigated, and the size range (in terms of KLOC and # of classes). Among the analyzed projects we have:

Table 7.1: Characteristics of the software systems used in the study.

Project	Period	#Classes	KLOC
Apache Ant	Jan 2000-Jan 2013	44-1,224	8-220
Apache Tomcat	Mar 2006-Jan 2013	828-1,548	254-350
jEdit	Sep 2001-July 2010	279-544	85-175
Android API (framework-opt-telephony)	Aug 2011-Jan 2013	218-225	73-78
Android API (frameworks-base)	Oct 2008-Jan 2013	1,698-3,710	534-1,043
Android API (frameworks-support)	Feb 2011-Nov 2012	199-256	58-61
Android API (sdk)	Oct 2008-Jan 2013	132-315	14-82
Android API (tool-base)	Nov 2012-Jan 2013	471-714	80-134
Apache Commons Lang	Jul 2002-Oct 2013	30-242	14-165
Apache Cassandra	Mar 2009-Oct 2013	313-1,008	115-935
Apache Commons Codec	Apr 2004-Jul 2013	23-107	4-25
Apache Derby	Aug 2008-Oct 2013	1,298-2,847	159-179
Eclipse Core	Jun 2001-Sep 2013	824-1,232	120-174
Apache James Mime4j	Jun 2005-Sep 2013	106-269	91-532
Google Guava	Sep 2009-Oct 2013	65-457	4-35
Aardvark	Nov 2010-Jan 2013	16-157	13-25
And Engine	Mar 2010-Jun 2013	215-613	14-24
Apache Commons IO	Jan 2002-Oct 2013	13-200	3-56
Apache Commons Logging	Aug 2001-Oct 2013	5-65	1-54
Mongo DB	Jan 2009-Oct 2013	13-27	10-25

- Nine projects coming from the Apache ecosystem[4], namely `Ant`, `Tomcat`, `Cassandra`, `Derby`, `James Mime4j`, and a set of `commons` libraries such as `Commons Lang`, `Commons Codec`, `Commons IO`, and `Commons Logging`.

- Five projects belonging to the Android APIs[5], namely `sdk`, `tool-base`, and a set of Android frameworks such as `opt-telephony`, `frameworks-base`, `frameworks-support`. Each of these projects is responsible for implementing parts of the Android APIs.

- Six open source projects from elsewhere: `jEdit`[6], `Eclipse Core`[7], `Google`

[4] `http://www.apache.org/` verified on September 2014
[5] `https://android.googlesource.com/` verified on September 2014
[6] `http://www.jedit.org/` verified on September 2014
[7] `http://www.eclipse.org/eclipse/platform-core/` verified on September 2014

`Guava`[8], `Aardvark`[9], `And Engine`[10], and `Mongo DB`[11].

Note that our choice of the subject systems is not random, but guided by specific requirements of our underlying infrastructure. Specifically, the selected systems:

1. are written in Java, since the MARKOS code analyzer is currently able to parse just systems written in this programming language;
2. have their entire development histories tracked in a versioning system;
3. have different development history lengths (we start with a minimum of three months for `tool-base` up to 13 years for `Apache Ant`); and
4. have different sizes (we go from a minimum of 25 KLOCs for `Commons Codec` up to 1,043 KLOCs for `framework-base`).

Research Questions

Our study aims at addressing the following two research questions:

- **RQ1:** *How does HIST perform in detecting code smells?* This research question aims at quantifying the accuracy of HIST in detecting instances of the five smells described in Section 7.2, namely *Divergent Change*, *Shotgun Surgery*, *Parallel Inheritance*, *Blob*, and *Feature Envy*.
- **RQ2:** *How does HIST compare to the smell detection techniques based on the analysis of a single project snapshot?* This research question aims at comparing the accuracy of HIST in detecting the five smells above with the accuracy achieved by applying a more conventional approach based on the analysis of a single project snapshot. The results of this comparison will provide insights into the usefulness of historical information while detecting smells.

[8] `https://code.google.com/p/guava-libraries/` verified on September 2014
[9] `http://karmatics.com/aardvark/` verified on September 2014
[10] `http://www.andengine.org/` verified on September 2014
[11] `http://www.mongodb.org/` verified on September 2014

Table 7.2: Snapshots considered for the smell detection.

Project	git snapshot	Date	Classes	KLOC
Apache Ant	da641025	Jun 2006	846	173
Apache Tomcat	398ca7ee	Jun 2010	1,284	336
jEdit	feb608el	Aug 2005	316	101
Android API (framework-opt-telephony)	b3a03455	Feb 2012	223	75
Android API (frameworks-base)	b4ff35df	Nov 2011	2,766	770
Android API (frameworks-support)	0f6f72e1	Jun 2012	246	59
Android API (sdk)	6feca9ac	Nov 2011	268	54
Android API (tool-base)	cfebaa9b	Dec 2012	532	119
Apache Commons Lang	4af8bf41	Jul 2009	233	76
Apache Cassandra	4f9e551	Sep 2011	826	117
Apache Commons Codec	c6c8ae7a	Jul 2007	103	23
Apache Derby	562a9252	Jun 2006	1,746	166
Eclipse Core	0eb04df7	Dec 2004	1,190	162
Apache James Mime4j	f4ad2176	Mar 2009	250	280
Google Guava	e8959ed0	Aug 2012	153	16
Aardvark	ff98d508	Jun 2012	103	25
And Engine	f25236e4	Oct 2011	596	20
Apache Commons IO	c8cb451c	Oct 2010	108	27
Apache Commons Logging	d821ed3e	May 2005	61	23
Mongo DB	b67c0c43	Oct 2011	22	25

Study Procedure, Data Analysis and Metrics

In order to answer **RQ1** we simulated the use of HIST in a realistic usage scenario. In particular, we split the history of the twenty subject systems into two equal parts, and ran our tool on all snapshots of the first part. For instance, given the history of `Apache Ant` going from January 2000 to January 2013, we selected a system snapshot s from June 2006. Then, HIST analyzed all snapshots from January 2000 to June 2006 in order to detect smell instances on the selected snapshot s. This was done aiming at simulating a developer performing smell detection on an evolving software system. On the one hand, considering some early snapshot in the project history, there could have been the risk of performing smell detection on a software system still exhibiting some ongoing, unstable design decisions. On the other hand, by considering snapshots occurring later in the project history (*e.g.*, the last available release) there could have been the risk of simulating some unrealistic scenario, *i.e.*, in which developers put effort in improving the design of a

software system when its development is almost absent. Table 7.2 reports the list of selected snapshots, together with their characteristics.

To evaluate the detection accuracy of HIST, we need an oracle reporting the instances of smells in the considered systems' snapshots. Unfortunately, there are no annotated sets of such smells available in literature. Thus, we had to manually build our own oracle. A Master's student from the University of Salerno manually identified instances of the five considered smells in each of the systems' snapshots. Starting from the definition of the five smells reported in literature, the student manually analyzed the source code of each snapshot, looking for instances of those smells. Clearly, for smells having an intrinsic historical nature, he analyzed the changes performed by developers on different code components. This process took four weeks of work. Then, a second Master's student (still from the University of Salerno) validated the produced oracle, to verify that all affected code components identified by the first student were correct. Only six of the smells identified by the first student were classified as false positives by the second student. After a discussion performed between the two students, two of these six smells were classified as false positives (and thus removed from the oracle). Note that, while this does not ensure that the defined oracle is complete (*i.e.*, it includes all affected components in the systems), it increases our degree of confidence on the correctness of the identified smell instances. To avoid any bias in the experiment, students were not aware of the experimental goals and of specific algorithms used by HIST for identifying smells. The number of code smell instances in our oracle is shown in Table 7.3 for each of the twenty subject systems. As we can see *Parallel Inheritance*, *Blob*, and *Feature Envy* code smells are quite diffused, presenting more than 50 instances each. A high number (24) of *Divergent Change* instances is also present in our oracle, while the *Shotgun Surgery* smell seems to be poorly diffused across open source projects, with just six instances identified.

Once we defined the oracle and obtained the set of smells detected by HIST on each of the systems' snapshots, we evaluated its detection accuracy by using two widely-adopted Information Retrieval (IR) metrics, namely precision and recall [55]:

Table 7.3: Code smell instances in the manually defined oracle.

Project	Divergent Change	Shotgun Surgery	Parallel Inheritance	Blob	Feature Envy
Apache Ant	0	0	7	8	8
Apache Tomcat	5	1	9	5	3
jEdit	4	1	3	5	10
Android API (framework-opt-telephony)	0	0	0	13	0
Android API (frameworks-base)	3	1	3	18	17
Android API (frameworks-support)	1	1	0	5	0
Android API (sdk)	1	0	9	10	3
Android API (tool-base)	0	0	0	0	0
Apache Commons Lang	1	0	6	3	1
Apache Cassandra	3	0	3	2	28
Apache Commons Codec	0	0	0	1	0
Apache Derby	0	0	0	9	0
Eclipse Core	1	1	8	4	3
Apache James Mime4j	1	0	0	0	9
Google Guava	0	0	0	1	2
Aardvark	0	1	0	1	0
And Engine	0	0	0	0	1
Apache Commons IO	1	0	1	2	1
Apache Commons Logging	2	0	2	2	0
Mongo DB	1	0	0	3	0
Overall	**24**	**6**	**51**	**92**	**86**

$$precision = \frac{|correct \cap detected|}{|detected|} \qquad recall = \frac{|correct \cap detected|}{|correct|} \tag{7.3}$$

where $correct$ and $detected$ represent the set of true positive smells (those manually identified) and the set of smells detected by HIST, respectively. As an aggregate indicator of precision and recall, we report the F-measure, defined as the harmonic mean of precision and recall:

$$F\text{-}measure = 2 * \frac{precision * recall}{precision + recall}\% \tag{7.4}$$

Turning to **RQ2**, we executed smell detection techniques based on the analysis of a single snapshot on the same systems' snapshots previously selected when answering **RQ1**. To the best of our knowledge, there is not a single approach detecting

all the smells that we considered in our study. For this reason, depending on the specific smell being detected, we considered different competitive techniques to compare our approach against. As for the *Blob*, we compared HIST with DECOR, the detection technique proposed by Moha *et al.* [20]. Specifically, we implemented the detection rules used by DECOR for the detection of *Blob*. Such rules are available online[12]. For the *Feature Envy* we considered JDeodorant as a competitive technique [25], which is a publicly available Eclipse plug-in[13]. The approach implemented in JDeodorant analyzes all methods for a given system, and forms a set of candidate target classes where a method should be moved into. This set is obtained by examining the entities (*i.e.*, attributes and methods) that a method accesses from the other classes.

As for *Divergent Change* and *Shotgun Surgery*, we compared HIST against our implementation of the approach proposed by Rao and Raddy [79] that is purely based on structural information. This technique starts by building an $n \times n$ matrix (where n is the number of classes in the system under analysis), named Design Change Propagation Probability (DCPP). A generic entry A_{ij} in DCPP represents the probability that a change in the class i triggers a change to the class j. Such a probability is given by the $cdegree$ [80], *i.e.*, an indicator of the number of dependencies that class i has with a class j (note that $cdegree$ is not symmetric, *i.e.*, $A_{ij} \neq A_{ji}$). Once the DCPP matrix is built, a *Divergent Change* instance is detected if a column in the matrix (*i.e.*, a class) has several (more than λ) non-zero values (*i.e.*, the class has dependencies with several classes). The conjecture is that if a class depends on several other classes, it is likely that it implements different responsibilities divergently changing during time. Regarding the detection of the *Shotgun Surgery*, instances of such a smell are identified when a row in the matrix (*i.e.*, a class) contains several (more than η) non-zero values (*i.e.*, several classes have dependencies with the class). The conjecture is that changes to this class will trigger changes in classes depending on it. From now on we will refer to this technique as DCPP.

[12] `http://www.ptidej.net/research/designsmells/grammar/Blob.txt`

[13] `http://www.jdeodorant.com/` verified on September 2014

Concerning the *Parallel Inheritance* smell, we are not aware of publicly available techniques in the literature to detect it. Thus, in order to have a meaningful baseline for HIST, we implemented a detection algorithm based on the analysis of a single project snapshot. Note that this analysis was not intended to provide evidence that HIST is the best method for detecting *Parallel Inheritance* instances. Instead, the goal was to conduct an investigation into the actual effectiveness of historical information while detecting smells as compared to information extracted from a single project snapshot.

We detect classes affected by *Parallel Inheritance* as pairs of classes having (i) both a superclass and/or a subclass (*i.e.*, both belonging to a class hierarchy), and (ii) the same prefix in the class name. This detection algorithm (from now on coined as PICA) directly comes from the Fowler's definition of *Parallel Inheritance*: *"You can recognize this smell because the prefixes of the class names in one hierarchy are the same as the prefixes in another hierarchy"* [8].

To compare the performances of HIST against the competitive techniques described above, we used recall, precision, and F-measure. Moreover, to analyze the complementarity of static code information and historical information when performing smell detection, we computed the following overlap metrics:

$$\text{correct}_{HIST \cap SS} = \frac{|\text{correct}_{HIST} \cap \text{correct}_{SS}|}{|\text{correct}_{HIST} \cup \text{correct}_{SS}|}\% \tag{7.5}$$

$$\text{correct}_{HIST \setminus SS} = \frac{|\text{correct}_{HIST} \setminus \text{correct}_{SS}|}{|\text{correct}_{HIST} \cup \text{correct}_{SS}|}\% \tag{7.6}$$

$$\text{correct}_{SS \setminus HIST} = \frac{|\text{correct}_{SS} \setminus \text{correct}_{HIST}|}{|\text{correct}_{HIST} \cup \text{correct}_{SS}|}\% \tag{7.7}$$

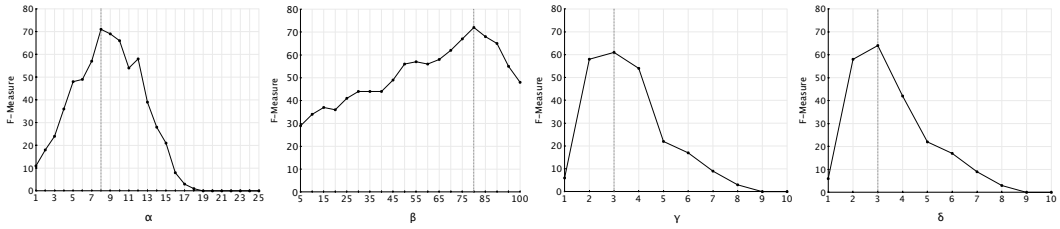
where $correct_{HIST}$ and $correct_{SS}$ represent the sets of correct smells detected by HIST and the competitive technique, respectively.

$correct_{HIST \cap SS}$ measures the overlap between the set of true smells detected by both techniques, and $correct_{HIST \setminus SS}$ ($correct_{SS \setminus HIST}$) measures the true smells detected by HIST (SS) only and missed by SS (HIST). The latter metric provides

an indication on how a smell detection strategy contributes to enriching the set of correct smells identified by another method.

Calibrating HIST and the Competitive Approaches

While for JDeodorant and DECOR parameter tuning has already been empirically investigated by their respective authors, we needed to calibrate parameters for HIST and DCPP as well. Indeed, in the work presenting the DCPP approach no best values for its parameters were recommended [79]. We performed this calibration on a software system which was not used in our experimentation, *i.e.*, `Apache Xerces`[14]. Also on this system, we asked two Master's students to manually identify instances of the five considered smells in order to build an oracle. The procedure adopted by the students was exactly the same described before and used to build the study oracle. Then, we evaluated the F-measure value obtained by the detection approaches using different settings.



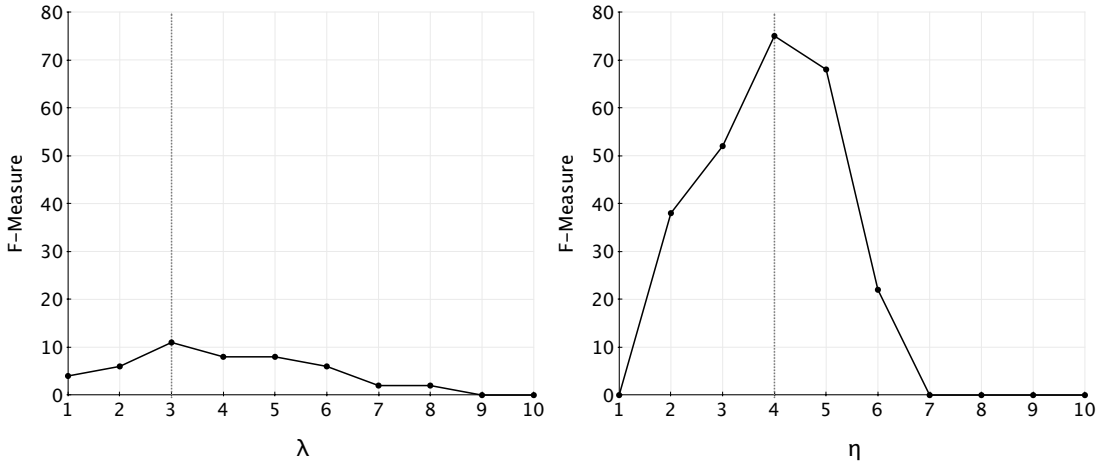
(a) Android (b) Apache (c) Eclipse (d) Eclipse

Figure 7.2: Parameters calibration for HIST (Blob) α (a), HIST (Feature Envy) β (b), HIST (Divergent Change) γ (c), and HIST (Shotgun Surgery) δ (d).

Results of the calibration are reported in Figure 7.2 for the HIST parameters α, β, γ, and δ, and in Figure 7.3 for the DCPP λ and the DCPP η parameters. As for the confidence and support, the calibration was not different from what was done in other work using association rule discovery [191, 192, 193, 194].

In particular, we tried all combinations of confidence and support obtained by varying the confidence between 0.60 and 0.90 by steps of 0.05, and the support

[14] `http://xerces.apache.org/` verified on September 2014



(a) Android (b) Apache

Figure 7.3: Parameters' calibration for DCPP-Divergent Change λ (a), and DCPP-Shotgun Surgery η (b).

between 0.004 and 0.04 by steps of 0.004, and searching for the one ensuring the best F-measure value on `Xerces` (that is the one that we used in answering the research questions).

Replication Package

The raw data and working data sets used in our study are publicly available in a replication package [195] where we provide: (i) links to the GIT repositories from which we extracted historical information; (ii) complete information on the change history in all the subject systems; (iii) the oracle used for each system; and (iv) the list of smells identified by HIST and by the competitive approaches.

7.3.2 Analysis of the Results

This section reports the results aimed at answering the two research questions formulated in Section 7.3.1. Note that to avoid redundancies, we report the results for both research questions together, discussing each smell separately.

Tables 7.4, 7.6, 7.7, 7.8, and 7.9 report the results—in terms of recall, preci-

Table 7.4: Divergent Change - HIST accuracy as compared to the single snapshot technique.

	#Smell	HIST						Single Snapshot technique					
Project	Instances	Identified	TP	FP	Prec.	Recall	F-measure	Identified	TP	FP	Prec.	Recall	F-measure
Apache Ant	0	0	0	0	-	-	-	1	0	1	-	-	-
Apache Tomcat	5	6	3	3	50%	60%	55%	0	0	0	N/A	N/A	N/A
jEdit	4	3	3	0	100%	75%	86%	1	1	0	100%	25%	40%
Android API (framework-opt-telephony)	0	0	0	0	-	-	-	0	0	0	-	-	-
Android API (frameworks-base)	3	3	3	0	100%	100%	100%	0	0	0	N/A	N/A	N/A
Android API (frameworks-support)	1	1	1	0	100%	100%	100%	2	0	2	0%	0%	0%
Android API (sdk)	1	1	1	0	100%	100%	100%	0	0	0	N/A	N/A	N/A
Android API (tool-base)	0	1	0	1	-	-	-	0	0	0	-	-	-
Apache Commons Lang	1	1	1	0	100%	100%	100%	0	0	0	N/A	N/A	N/A
Apache Cassandra	3	2	2	0	100%	67%	80%	7	1	6	14%	34%	20%
Apache Commons Codec	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Derby	0	0	0	0	-	-	-	0	0	0	-	-	-
Eclipse Core	1	1	1	0	100%	100%	100%	0	0	0	N/A	N/A	N/A
Apache James Mime4j	1	1	1	0	100%	100%	100%	1	1	0	100%	100%	100%
Google Guava	0	0	0	0	-	-	-	0	0	0	-	-	-
Aardvark	0	1	0	1	-	-	-	0	0	0	-	-	-
And Engine	0	0	0	0	-	-	-	14	0	14	-	-	-
Apache Commons IO	1	1	1	0	100%	100%	100%	3	0	3	0%	0%	0%
Apache Commons Logging	2	2	2	0	100%	100%	100%	0	0	0	N/A	N/A	N/A
Mongo DB	1	1	1	0	100%	100%	100%	0	0	0	N/A	N/A	N/A
Overall	**24**	**25**	**20**	**5**	**80%**	**83%**	**82%**	**29**	**3**	**26**	**10%**	**13%**	**11%**

sion, and F-measure—achieved by HIST and approaches based on the analysis of a single snapshot on the twenty subject systems. In addition, each table also reports (i) the number of smell instances present in each system (column "#Smell Instances"), (ii) the number of smell instances identified by each approach (column "Identified"), (iii) the number of true positive instances identified by each approach (column "TP"), and (iv) the number of false positive instances identified by each approach (column "FP"). Note that each table shows the results for one of the five smells considered in our study and in particular: Table 7.4 for *Divergent Change*, Table 7.6 for *Shotgun Surgery*, Table 7.7 for *Parallel Inheritance*, Table 7.8 for *Blob*, and Table 7.9 for *Feature Envy*.

As explained in Section 7.3.1 for *Divergent Change* and *Shotgun Surgery* we compared HIST against DCCP approach proposed by Rao and Raddy [79], while for *Parallel Inheritance* we used an alternative approach that we developed (PICA). Finally, for *Blob* and *Feature Envy* we used DECOR rules [20] and the JDeodorant tool [25], respectively.

When no instances of a particular smell were present in the oracle (i.e., zero in the column "#Smell Instances"), it was not possible to compute the recall (that is, division by zero), while the precision would be zero if at least one false positive is detected (independently of the number of false positives). In these cases a "-" is indicated in the corresponding project row. Similarly, when an approach did not retrieve any instances of a particular smell, it was not possible to compute precision, while recall would be zero if at least one false positive is retrieved. In this case a "N/A" is included in the project row. However, to have an accurate estimation of the performances of the experimented techniques, we also report in each table the results achieved by considering all systems as a single dataset (rows "Overall"). In such a dataset, it never happens that recall or precision cannot be computed for the reasons described above. Thus, all true positives and all false positives identified by each technique are taken into account in the computation of the overall recall, precision, and F-measure.

Finally, Table 7.5 reports the overlap and differences between HIST and the techniques based on code analysis of a single snapshot: column "HIST $\cap$ SS Tech." reports the number (#) and percentage (%) of smells correctly identified by both HIST and the competitive technique; column "HIST $\setminus$ SS Tech." reports the number and percentage of smells correctly identified by HIST but not by the competitive technique; column "SS Tech. $\setminus$ HIST" reports the number and percentage of smells correctly identified by the competitive technique but not by HIST. In the following, we discuss the results for each kind of smell.

Divergent Change

We identified 24 instances of *Divergent Change* in the twenty systems (see Table 7.4). The results clearly indicate that the use of historical information allows to outperform DCCP (*i.e.*, the approach based on the analysis of a single snapshot). Specifically, the F-measure achieved by HIST on the overall dataset is 82% (83% of recall and 80% of precision) against 10% (13% of recall and 11% of precision) achieved by DCCP. This is an expected result, since the *Divergent Change* is by definition a "historical smell" (see Section 7.2), and thus we expected difficulties

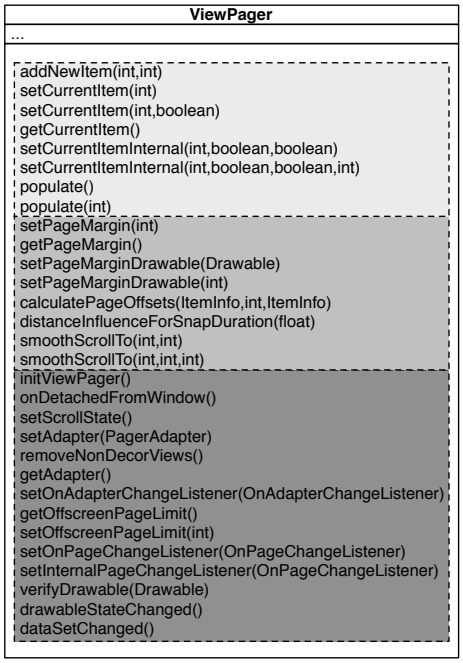


Figure 7.4: One of the identified *Divergent Change* instances: the class `ViewPager` from `Android frameworks-support`.

in capturing this kind of smell by just relying on the analysis of a single system's snapshot.

One of the *Divergent Change* instances captured by HIST is depicted in Figure 7.4 and related to the `ViewPager` class from the `Android frameworks-support` project. `ViewPager` allows users of Android apps to flip left and right through pages of data. In this class, HIST identified three sets of methods divergently changing during the project's history (see Section 7.2.2 for details on how these sets were identified). The three sets are highlighted in Figure 7.4 by using different shades of gray. Starting from the top of the Figure 7.4, the first set groups together methods somewhat related to the management of the `items` to be displayed in the `View` (*e.g.*, menu, buttons, *etc.*). The middle set gathers methods allowing to manage the `View` layout (*i.e.*, setting margins, page offsets, *etc.*), while the set at the bottom of Figure 7.6 is mainly related to the `View` configuration (*e.g.*, init the page viewer, define the change listeners, *etc.*). Thus, the three identified sets of methods, not only change independently one from the other, but also seem

Table 7.5: Overlap between HIST and Single Snapshot (SS) techniques. For Blob the SS Tech. is DECOR, for Feature Envy it is JDeodorant.

Code Smell	HIST∩SS Tech.		HIST\SS Tech.		SS Tech.\HIST	
	#	%	#	%	#	%
Divergent Change	1	4%	19	87%	2	9%
Shotgun Surgery	0	0%	6	100%	0	0%
Parallel Inheritance	20	50%	15	38%	5	12%
Blob	13	16%	40	51%	27	33%
Feature Envy	44	54%	22	27%	17	19%

to represent quite independent responsibilities implemented in the `ViewPager` class. Of course, no speculations can be made on the need for refactoring of this class, since developers having high experience on the system are needed to evaluate both pros and cons.

Going back to the quantitative results, DCCP was able to detect only three correct occurrences of *Divergent Change* and one of them was also captured by HIST. The instances missed by HIST (and identified by DCCP) affect the `RE` class of `jEdit` and the `CassandraServer` class of `Apache Cassandra`. Both of these classes do not have enough change history data about divergent changes to be captured by HIST. This clearly highlights the main limitation of HIST that requires sufficient amount of historical information to infer useful association rules.

Given these observations, the overlap between the smells detected by HIST and DCCP results reported in Table 7.5 is quite expected: among the sets of smells correctly detected by two techniques, there is just a 4% overlap, HIST is the only one retrieving 87% of the smells, while DCCP is the one detecting only two smells described above and missed by HIST (9%). Thus, the complementarity between HIST and DCCP is rather low.

Shotgun Surgery

Shotgun Surgery is the smell with the lowest number of instances in the subject systems, *i.e.*, with only six systems affected for a total of six instances (one per system).

Table 7.6: Shotgun Surgery - HIST accuracy compared to the single snapshot techniques.

	#Smell	HIST						Single snapshot echnique					
Project	Instances	Identified	TP	FP	Prec.	Recall	F-measure	Identified	TP	FP	Prec.	Recall	F-measure
ArgoUML	0	0	0	0	-	-	-	4	0	4	-	-	-
Apache Ant	1	1	1	0	100%	100%	100%	13	0	13	0%	0%	0%
jEdit	1	1	1	0	100%	100%	100%	3	0	3	0%	0%	0%
Android API (framework-opt-telephony)	0	1	0	1	-	-	-	3	0	3	-	-	-
Android API (frameworks-base)	1	1	1	0	100%	100%	100%	1	0	1	0%	0%	0%
Android API (frameworks-support)	1	1	1	0	100%	100%	100%	2	0	2	0%	0%	0%
Android API (sdk)	0	0	0	0	-	-	-	0	0	0	-	-	-
Android API (tool-base)	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Commons Lang	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Cassandra	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Commons Codec	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Derby	0	0	0	0	-	-	-	0	0	0	-	-	-
Eclipse Core	1	1	1	0	100%	100%	100%	0	0	0	N/A	N/A	N/A
Apache James Mime4j	0	0	0	0	-	-	-	0	0	0	-	-	-
Google Guava	0	0	0	0	-	-	-	0	0	0	-	-	-
Aardvark	1	1	1	0	100%	100%	100%	0	0	0	N/A	N/A	N/A
And Engine	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Commons IO	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Commons Logging	0	0	0	0	-	-	-	0	0	0	-	-	-
Mongo DB	0	0	0	0	-	-	-	0	0	0	-	-	-
Overall	**6**	**7**	**6**	**1**	**86%**	**100%**	**92%**	**26**	**0**	**26**	**0%**	**0%**	**0%**

HIST was able to detect all the instances of this smell (100% recall) with 86% precision, outperforming DCCP (*i.e.*, the competitive approach). Specifically, DCCP was not able to detect any of the six instances of this smell present in the subject systems. Thus, no meaningful observations can be made in terms of overlap metrics. This result highlights the fact that it is quite difficult to identify characteristics of such a smell by solely analysing a single system's snapshot, as the smell is intrinsically defined in terms of a change triggering many other changes [8].

It is also worthwhile to discuss an example of *Shotgun Surgery* we identified in `Apache Tomcat` and represented by the method `isAsync` of the class named `AsyncStateMachine`. HIST identified association rules between this method and 48 methods in the system, belonging to 31 different classes. This means that, whenever the `isAsync` method is modified, also these 48 methods, generally, undergo a change. Figure 7.5 shows all 31 classes involved: each arrow going from the `isAsync` method to one of these 31 classes is labeled with the number of times `isAsync` co-changed with methods of that class in the analyzed time pe-

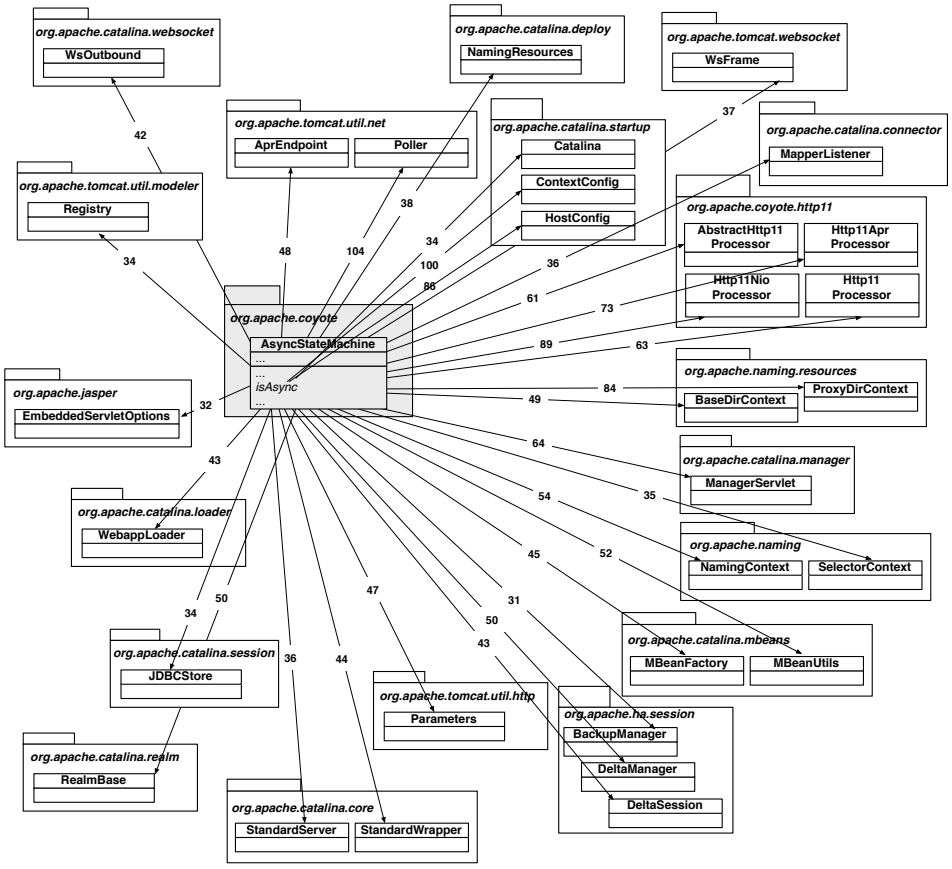


Figure 7.5: One of the identified *Shotgun Surgery* instances: the `AsyncStateMachine.isAsync` method from `Apache Tomcat`.

riod. Note that the total number of changes performed in the analyzed time period to `isAsync` is 110. For instance, `isAsync` co-changed 104 (95%) times with two methods contained in the `Poller` class. What is also very surprising about this instance of *Shotgun Surgery* is that it triggers changes in over 19 different packages of the software system. This clearly highlights the fact that such smell could be very detrimental in software evolution and maintenance context.

As for the only false positive instance identified by HIST, it is the method `dispose` of the class `AsyncStateMachine` contained in the Android library `framework-opt-telephony` (see Table 7.6). HIST identified association rules between this method and three other methods in the system: the method `dispose` of the class `CdmaDataConnectionTracker`, the method `handleSendComplete`

Table 7.7: Parallel Inheritance - HIST accuracy as compared to the single snapshot techniques.

	#Smell	HIST						Single snapshot technique					
Project	Instances	Identified	TP	FP	Prec.	Recall	F-measure	Identified	TP	FP	Prec.	Recall	F-measure
Apache Ant	7	8	5	3	63%	71%	67%	52	4	48	8%	57%	14%
Apache Tomcat	9	10	6	4	60%	67%	63%	61	4	57	7%	44%	12%
jEdit	3	0	0	0	N/A	N/A	N/A	15	3	12	20%	100%	33%
Android API (framework-opt-telephony)	0	0	0	0	-	-	-	9	0	9	-	-	-
Android API (frameworks-base)	3	1	0	1	0%	0%	0%	111	0	111	0%	0%	0%
Android API (frameworks-support)	0	0	0	0	-	-	-	9	0	9	-	-	-
Android API (sdk)	9	12	8	4	67%	89%	76%	59	3	56	5%	33%	12%
Android API (tool-base)	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Commons Lang	6	6	6	0	100%	100%	100%	6	6	0	100%	100%	100%
Apache Cassandra	3	1	1	0	100%	34%	50%	35	1	34	3%	34%	5%
Apache Commons Codec	0	0	0	0	-	-	-	3	0	3	-	-	-
Apache Derby	0	0	0	0	-	-	-	53	0	53	-	-	-
Eclipse Core	8	8	7	1	88%	88%	88%	31	2	29	6%	25%	10%
Apache James Mime4j	0	0	0	0	-	-	-	10	0	10	-	-	-
Google Guava	0	0	0	0	-	-	-	0	0	0	-	-	-
Aardvark	0	0	0	0	-	-	-	0	0	0	-	-	-
And Engine	0	0	0	0	-	-	-	60	0	60	-	-	-
Apache Commons IO	1	1	1	0	100%	100%	100%	8	1	7	13%	100%	22%
Apache Commons Logging	2	1	1	0	100%	50%	67%	3	1	2	34%	50%	40%
Mongo DB	0	0	0	0	-	-	-	0	0	0	-	-	-
Overall	**51**	**48**	**35**	**13**	**73%**	**69%**	**71%**	**525**	**25**	**500**	**5%**	**49%**	**9%**

of the class `SMSDispatcher`, and the method `dump` of the class `GsmCallTracker`. However, this behavior was not considered as "smelly" by the students building the oracle because: (i) differently from what discussed for the `isAsync` method, the triggered changes in this case are spread just across three classes, and (ii) even if the four involved methods tend to change together, they are correctly placed into different classes splitting well the system's responsibilities. For instance, while the two `dispose` methods are both in charge of cleaning up a data connection, the two protocols they manage are different (*i.e.*, GSM *vs* CDMA). Thus, even if they co-change during time, there is no apparent reason for placing them in the same class with the only goal of isolating the change (poorly spread in this case). Indeed, as a side effect, this refactoring operation could create a class managing heterogeneous responsibilities (*i.e.*, a *Blob* class).

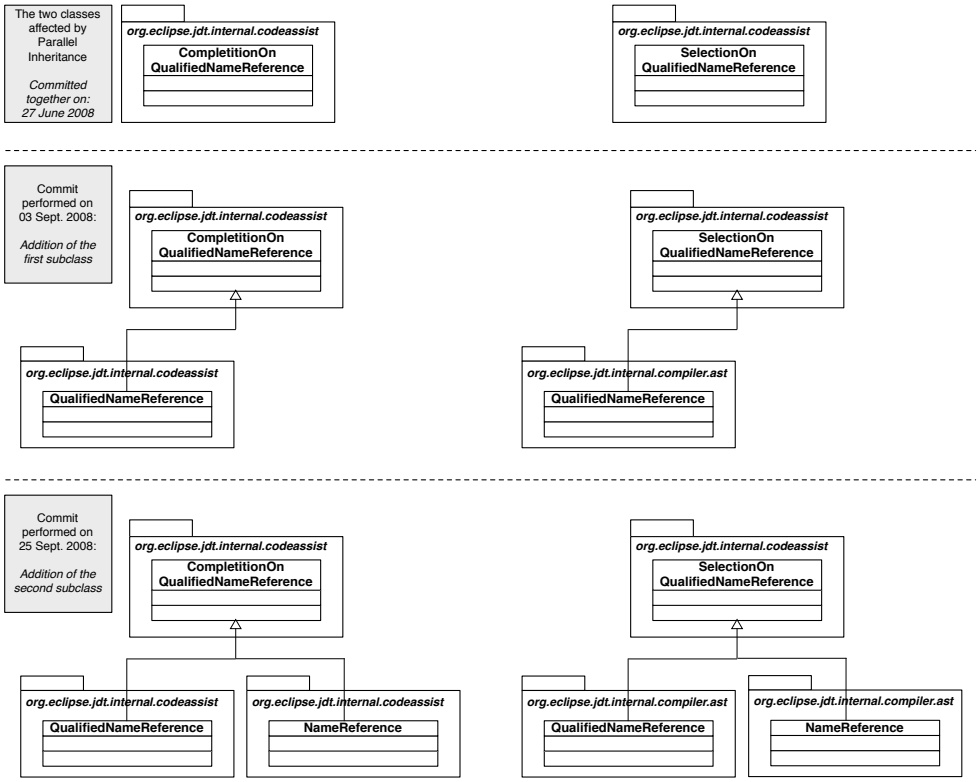


Figure 7.6: `CompletitionOnQualifiedNameReference` and `SelectionOnQualifiedNameReference` from `Eclipse JDT`, identified as *Parallel Inheritance*.

Parallel Inheritance

Among the 51 instances of the *Parallel Inheritance* smell, HIST was able to correctly identify 35 of them (recall 69%) with a price to pay of 13 false positives, resulting in a precision of 73%. By using the competitive technique (*i.e.,* PICA) we were able to retrieve 25 correct instances of the smell (recall of 49%) while also retrieving 473 false positives (precision of 5%).

One of the *Parallel Inheritance* instances detected by HIST and missed by the alternative approach PICA is depicted in Figure 7.6. The example refers to two classes of the `Eclipse JDT`, *i.e.*, `CompletitionOnQualifiedNameReference` and the class named `SelectionOnQualifiedNameReference`. As shown in Figure 7.6, these two classes have been committed together on 27 June 2008 in

the same package and since then, the hierarchies having them as top superclasses evolved in parallel. Indeed, the first subclass (`QualifiedNameReference`) has been added to both superclasses on 3 September 2008 followed by the second subclass (`NameReference`) on 25 September 2008. Note that while the name of the subclasses added to the two superclasses is the same, we are talking about two different subclasses. Indeed, as show in Figure 7.6, these subclasses are from different packages. For instance, the `NameReference` class, which is a subclass of the class `CompletitionOnQualifiedNameReference`, is from the package named `internal.codeassist`, while the corresponding subclass of the class `SelectionOnQualifiedNameReference` is from `internal.compiler`.

Looking at the overlap metrics reported in Table 7.5, we can see an overlap of 50% among the set of smells correctly identified by the two techniques, while 38% of the correct instances are retrieved only by HIST and the remaining 12% are identified only by PICA. For example, an instance of *Parallel Inheritance* detected by PICA and missed by HIST is the one affecting `Broken2OperationEnum` and `Broken5OperationEnum` belonging to `Apache Commons Lang`. In this case, while the two hierarchies co-evolved synchronously, the (too high) thresholds used for the support and confidence of the association rule mining algorithm used in HIST did not allow capturing this specific instance (and thus, to identify the smell). Obviously, this instance could have been detected when using lower values for support and confidence, however, this would naturally result in drastically decreasing precision while somewhat increasing recall values.

Blob

As for detecting the *Blobs*, HIST was able to achieve a precision of 72% and a recall of 58% (F-measure=64%), while DECOR was able to achieve a precision of 54% and a recall of 43% (F-measure=48%). In more details, HIST achieved better precision values on 13 systems (on average, +45%), DECOR on two systems (on average, +45%), while on one system there was a tie. Thus, for most of the systems containing *Blob* instances (13 out of 16) HIST requires less effort to developers looking for instances of Blobs due to the lower number of false positives

Table 7.8: Blob - HIST accuracy as compared to DECOR.

	#Smell	HIST						DECOR					
Project	Instances	Identified	TP	FP	Prec.	Recall	F-measure	Identified	TP	FP	Prec.	Recall	F-measure
Apache Ant	8	10	6	4	60%	75%	67%	10	3	7	30%	38%	33%
Apache Tomcat	5	1	1	0	100%	20%	33%	6	4	2	67%	80%	73%
jEdit	5	3	2	1	67%	40%	50%	5	3	2	60%	60%	60%
Android API (framework-opt-telephony)	13	10	10	0	100%	77%	87%	10	7	3	70%	54%	61%
Android API (frameworks-base)	18	13	9	4	70%	50%	58%	14	9	5	65%	50%	57%
Android API (frameworks-support)	5	7	5	2	71%	100%	83%	8	3	5	38%	60%	49%
Android API (sdk)	10	7	6	1	86%	60%	71%	7	2	5	29%	20%	24%
Android API (tool-base)	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Commons Lang	3	2	2	0	100%	67%	80%	0	0	0	N/A	N/A	N/A
Apache Cassandra	2	0	0	0	N/A	N/A	N/A	0	0	0	N/A	N/A	N/A
Apache Commons Codec	1	2	1	1	50%	100%	67%	0	0	0	N/A	N/A	N/A
Apache Derby	9	0	0	0	N/A	N/A	N/A	7	4	3	57%	44%	50%
Eclipse Core	4	3	2	1	67%	50%	57%	4	2	2	50%	50%	50%
Apache James Mime4j	0	3	0	3	-	-	-	0	0	0	-	-	-
Google Guava	1	1	1	0	100%	100%	100%	0	0	0	N/A	N/A	N/A
Aardvark	1	1	1	0	100%	100%	100%	1	1	0	100%	100%	100%
And Engine	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Commons IO	2	3	2	1	67%	100%	80%	0	0	0	N/A	N/A	N/A
Apache Commons Logging	2	3	2	1	67%	100%	80%	2	2	0	100%	100%	100%
Mongo DB	3	5	3	2	60%	100%	75%	0	0	0	N/A	N/A	N/A
Overall	**92**	**74**	**53**	**21**	**72%**	**58%**	**64%**	**74**	**40**	**34**	**54%**	**43%**	**48%**

that will be inspected and discarded. Also, HIST ensured better recall on nine out of the 16 systems containing at least one *Blob* class, and a tie has been reached on five other systems. On the contrary, HIST outperformed by DECOR on `Apache Tomcat` and `jEdit` (see Table 7.8). However, on the overall dataset, HIST was able to correctly identify 53 of the 92 existing Blobs, against the 40 identified by DECOR. Thus, as also indicated by the F-measure value computed over the whole dataset, the overall performance of HIST is better than that one of DECOR (64% against 48%). Noticeably, the two approaches seem to be highly complementary. This is highlighted by the overlap results in Table 7.5. Among the sets of smells correctly identified by the two techniques, there is an overlap of just 16%. Specifically, HIST is able to detect 51% of smells that are ignored by DECOR, and the latter retrieves 33% of correct smells that are not identified by HIST. Similarly to the results for the *Parallel Inheritance* smell, this finding highlights the possibility of building better detection techniques by combining single-snapshot code analysis and change history information.

An example of *Blob* correctly identified by HIST and missed by DECOR is the

Table 7.9: Feature Envy - HIST accuracy as compared to JDeodorant.

	#Smell	HIST						JDeodorant					
Project	Instances	Identified	TP	FP	Prec.	Recall	F-measure	Identified	TP	FP	Prec.	Recall	F-measure
Apache Ant	8	9	6	3	67%	75%	71%	13	2	11	15%	25%	19%
Apache Tomcat	3	1	1	0	100%	33%	50%	3	2	1	67%	67%	67%
jEdit	10	10	8	2	100%	100%	100%	3	3	0	100%	27%	43%
Android API (framework-opt-telephony)	0	0	0	0	-	-	-	0	0	0	-	-	-
Android API (frameworks-base)	17	24	15	9	63%	88%	73%	16	16	0	100%	94%	96%
Android API (frameworks-support)	0	0	0	0	-	-	-	0	0	0	-	-	-
Android API (sdk)	3	1	1	0	100%	33%	50%	0	0	0	N/A	N/A	N/A
Android API (tool-base)	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Commons Lang	1	2	1	1	50%	100%	67%	2	1	1	50%	100%	67%
Apache Cassandra	28	28	28	0	100%	100%	100%	28	28	0	100%	100%	100%
Apache Commons Codec	0	1	0	1	-	-	-	0	0	0	-	-	-
Apache Derby	0	0	0	0	-	-	-	0	0	0	-	-	-
Eclipse Core	3	5	3	2	60%	100%	75%	0	0	0	N/A	N/A	N/A
Apache James Mime4j	9	0	0	0	N/A	N/A	N/A	11	9	2	82%	100%	90%
Google Guava	2	2	2	0	100%	100%	100%	3	0	3	0%	0%	0%
Aardvark	0	0	0	0	-	-	-	0	0	0	-	-	-
And Engine	1	2	1	1	50%	100%	67%	0	0	0	N/A	N/A	N/A
Apache Commons IO	1	0	0	0	N/A	N/A	N/A	6	0	6	0%	0%	0%
Apache Commons Logging	0	0	0	0	-	-	-	8	0	8	-	-	-
Mongo DB	0	0	0	0	-	-	-	1	0	1	-	-	-
Overall	**86**	**85**	**66**	**19**	**78%**	**77%**	**77%**	**94**	**61**	**33**	**65%**	**71%**	**68%**

class `ELParser` from `Apache Tomcat`, that underwent changes in 178 out of the 1,976 commits occurred in the analyzed time period. `ELParser` is not retrieved by DECOR because this class has a one-to-one relationship with data classes, while a one-to-many relationship is required by the DECOR detection rule. Instead, a *Blob* retrieved by DECOR and missed by HIST is the class `StandardContext` of `Apache Tomcat`. While this class exhibits all the structural characteristics of a *Blob* (thus allowing DECOR to detect it), it was not involved in any of the commits (*i.e.*, it was just added and never modified), hence making the detection impossible for HIST.

Feature Envy

For the *Feature Envy*, we found instances of this smell in twelve out of the twenty systems, for a total of 86 affected methods. HIST was able to identify 66 of them (recall of 77%) against the 61 identified by JDeodorant (recall of 71%). Also, the precision obtained by HIST is higher than the one achieved by JDeodorant (78% against 65%). However, it is important to point out that JDeodorant is a refactoring

tool and, as such, it identifies *Feature Envy* smells in software systems with the sole purpose of suggesting move method refactoring opportunities. Thus, the tool reports the presence of *Feature Envy* smells only if the move method refactoring is possible, by checking some preconditions ensuring that a program's behavior does not change after applying the suggested refactoring operation [25]. An example of considered preconditions is that *the envied class does not contain a method having the same signature as the moved method* [25]. To perform a fair comparison (especially in terms of recall), we filtered the *Feature Envy* instances retrieved by HIST by using the same set of preconditions defined by JDeodorant [25]. This resulted in the removal of three correct instances, as well as three false positives previously retrieved by HIST, thus decreasing the recall from 78% to 74% and increasing the precision from 78% to 80%. Still, HIST achieves better recall and precision values as compared to JDeodorant.

It is interesting to observe that the overlap data reported in Table 7.5 highlights, also in this case, some complementarity between historical and single snapshot techniques, with 54% of correct smell instances identified by both techniques (overlap), 27% identified only by HIST, and 19% only by JDeodorant.

An example of correct smell instance identified by HIST only is represented by the method `buildInputMethodListLocked` (`InputMethodManagerService` of the `Android framework-base` API). HIST identified the envied class in the `WindowManagerService` class, since there are just three commits in which the method is co-changed with methods of its class, against the 16 commits in which it is co-changed together with methods belonging to the envied class. Instead, JDeodorant was the only technique able to correctly identify the *Feature Envy* smell present in `Apache Ant`, affecting the method `isRebuildRequired` of the class `WebsphereDeploymentTool`. In this case, the envied class is `Project`, and HIST was not able to identify it due to the limited number of observed co-changes.

7.4 Threats to Validity

This section discusses the threats that could affect the validity of the evaluation of HIST.

Construct Validity Threats to *construct validity* concern relationships between theory and observation. This threat is generally due to imprecision in the measurements performed in the study. In the context of the study, this is mainly due to how the oracle was built (see Section 7.3.1). It is important to remark that to mitigate the bias for such a task, the students who defined the oracle were not aware of how HIST actually works. However, we cannot exclude that such manual analysis could have potentially missed some smells, or else identified some false positives. Another threat is due to the baselines—*i.e.*, competitive approaches—against which we compared HIST. While for *Blob*, *Feature Envy*, *Divergent Change*, and *Shotgun Surgery* we compared HIST against existing techniques/tools, this was not possible for the *Parallel Inheritance* smell, for which we had to define an alternative static detection technique, that may or may not be the most suitable ones among those based solely on structural information. Last, but not least, note that although we implemented the DECOR rules (for the *Blob* detection) and the approach by Rao *et al.* [79] (for *Divergent Change* and *Shotgun Surgery*) ourselves, these are precisely defined by the authors.

Internal Validity Threats to *internal validity* concern factors that could have influenced our results. In our context, a possible threat is represented by the calibration of the HIST parameters, as well as of those of the alternative static approaches. We performed the calibration of these parameters on one project (`Apache Xerces`) not used in our study, by computing F-measures for different possible values of such parameters (see Section 7.3.1).

External Validity Threats to *external validity* concern the generalization of the results. HIST only deals with five smells, while there might be many more left uncovered [8, 61]. However, as explained in Section 7.2 we focused on (i) three smells—*Divergent Change*, *Shotgun Surgery*, and *Parallel Inheritance*—that are clearly

related to how source code elements evolve over time, rather than to their structural characteristics, and (ii) two smells—*Blob* and *Feature Envy*—whose characteristics can be captured, at least in part, by observing source code changes over time. However, we cannot exclude that there could be other smells that can be modeled similarly.

We conducted it on twenty Java projects ensuring a good generalization of our findings. We evaluated HIST and the competitive techniques on a specific system's snapshot selected by splitting the history of each object system in two equal parts. Thus, the achieved results, and in particular our main finding in the context of **RQ2** (*i.e., HIST was able to outperform single snapshot techniques and tools in terms of recall, precision, and F-measure*), might be influenced by the specific selected snapshot. To mitigate such a threat, we replicated the study on ten snapshots representing ten different releases of a single system, namely `Apache Cassandra`. In particular, we considered `Cassandra`'s releases from 0.5 to 1.1[15]. Note that we just performed this analysis on a single system since it required the manual definition of ten new oracles (*i.e.,* one for each release) reporting the smell instances present in each release. The oracle definition was performed by two Master's students (one of which was also involved in the definition of the 20 oracles exploited in Study I) by adopting the same procedure described in Section 7.3.1. We run HIST and the competitive techniques on the ten snapshots representing the ten releases. Such snapshots have been identified by exploiting the `Git` tagging mechanism. The results achieved are high consistent when comparing HIST and the competitive techniques. Figure 7.7 reports the boxplots of the F-Measure achieved by HIST and by the competitive techniques on the ten `Cassandra` releases for each of the five considered code smells. The achieved results can be summarized as follows:

- *Divergent Change*: HIST achieves a higher F-Measure with respect to the competitive technique (*i.e.,* DCPP) in nine out of the ten considered releases (all but `Cassandra` 0.5).

[15]We discarded the first four releases (*i.e.,* from release 0.1 to release 0.4) since change-history information for these four releases was not present in the versioning system.

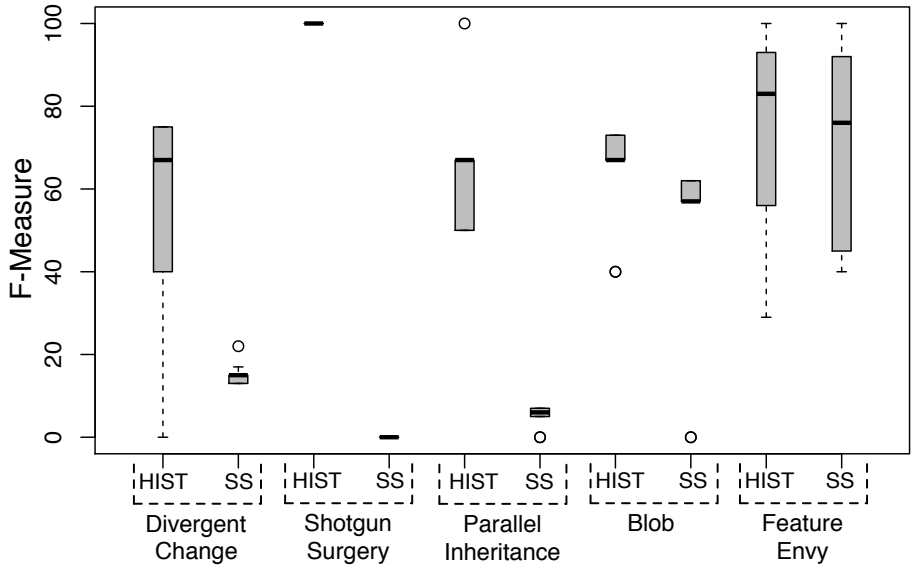


Figure 7.7: HIST *vs* single-snapshot competitive techniques (SS): F-Measure achieved for each smell type on the ten Cassandra releases.

- *Shotgun Surgery*: HIST achieves a higher F-Measure with respect to the competitive technique (*i.e.,* DCPP) in nine out of the ten considered releases (all but `Cassandra` 0.5). In `Cassandra` 0.5, a tie is reached, since no instances of the *Shotgun Surgery* smell are present, and both detection techniques do not retrieve any false positive.

- *Parallel Inheritance*: HIST achieves a higher F-Measure with respect to the competitive technique (*i.e.,* PICA) in all ten considered releases.

- *Feature Envy*: HIST achieves a higher F-Measure with respect to the competitive technique (*i.e.,* JDeodorant) in six out of the ten considered releases, JDeodorant works better on two releases (the first two, `Cassandra` 0.5 and 0.6), while a tie is reached on the remaining two releases.

- *Blob*: HIST outperforms the competitive static technique (*i.e.,* DECOR) in all ten considered releases.

Interestingly, when the competitive techniques outperform HIST, the releases involved are the 0.5 (in case of *Divergent Change*, *Shotgun Surgery*, and *Feature Envy*)

and the 0.6 (in case of *Feature Envy*), representing the first two considered releases. Thus, we can conclude that a shorter change history penalizes HIST as compared to the competitive techniques. Such a limitation is typical of all approaches exploiting historical information to derive recurring patterns. Despite that, the overall results achieved on the release-snapshots confirm our main finding reported while answering **RQ2**: HIST outperforms the competitive detection techniques based on code analysis of a single system snapshot. The interested reader can find detailed results about this analysis in our replication package [195].

Despite the effort we put in extending our evaluation to a high number of systems, it could be worthwhile to replicate the evaluation on other projects having different evolution histories or different architectures (*e.g.,* plugin-based architecture). Also, the number of code smell instances present in our oracle was quite low for the *Shotgun Surgery* smell (six instances). However, while this means evaluating the HIST performances on a small number of "true positive" instances, it is worth noting that achieving high precision levels is even harder when the number of correct instances in the oracle is low. Indeed, it is easier to identify a high number of false positives when the true positives in the oracle are very few. Despite this, HIST achieved an average precision of 86% for such a smell.

7.5 Conclusion

We presented HIST, an approach aimed at detecting five different code bad smells by exploiting co-changes extracted from versioning systems. We identified five smells for which historical analysis can be helpful in the detection process: *Divergent Change*, *Shotgun Surgery*, *Parallel Inheritance*, *Blob*, and *Feature Envy*. For each smell we defined a historical detector, using association rule discovery [190] or analyzing the set of classes/methods co-changed with the suspected smell.

We assessed HIST's recall and precision over a manually-built oracle of smells identified in twenty Java open source projects, and compared it with alternative smell detection approaches based on the analysis of a single project snapshot. The results of our study indicate that HIST exhibits a precision between 72% and 86%,

and a recall between 58% and 100%. For "intrinsically historical" smells—such as *Divergent Change*, *Shotgun Surgery*, *Parallel Inheritance*—HIST clearly outperforms approaches based on the analysis of a single snapshot, and generally performs as well these latter (if not better) for *Blob* and *Feature Envy* smells. Besides the better detection accuracy (in terms of precision and recall), HIST has a further advantage: it highlights smells that are subject to frequent changes, and therefore be possibly more problematic for the maintainer. In other words, a *Blob* detected based on structural information might not be necessarily a problem if it rarely (or never) changes, whereas it is worthwhile to bring to the attention of the developer those changing very frequently, hence identified by HIST. Finally, it is important to remark that in several cases the sets of smells detected by HIST and by techniques analyzing a single system's snapshot are quite complementary, suggesting that better techniques can be built by combining them.

This result triggers our future research agenda, aimed at developing a hybrid smell detection approach, obtained by combining static code analysis with analysis of co-changes. Also, we are planning on investigating the applicability of HIST to other types of smells. Finally, we would like to perform a deeper investigation into the characteristics causing a smell instance to represent/not represent a problem for developers.

Chapter 8

The Perception of Historical and Structural Smells

8.1 Introduction

Bad code smells have been defined by Martin Fowler as symptoms of the presence of poor design or implementation choices applied by programmers during the development of a software system [8]. Code smells attracted the attention of several researchers since has been demonstrated how bad design solutions impact the comprehensibility [18] and the maintenability of source code [16, 17, 19]. To the lights of such findings, approaches and tools for the automatic identification of the source code affected by code smells have been proposed. The vast majority of them are based on structural analysis, namely they analyze the structural characteristics of the source code (*e.g.,* method calls) in order to extract relevant informations that can be used as indicators of the presence of smells. For instance, the technique proposed by Moha *et al.* [20] combines a set of CK metrics [64], such as LCOM5 and the number of methods belonging to a class to estimate the smelliness of a code element. On the other hand, structural techniques are not suitable for the detection of several code smells presented in the catalogue by Fowler [8]. For this reason, alternative approaches have been devised [52, 53]. They use different types of sources of information, such as historical and textual analysis, to

evaluate whether a class/method is smelly. Specifically, the historical approach presented in Chapter 7, named HIST, identifies code smells relying on association rule discovery and change frequency analysis. The textual-based technique that is presented in Chapter 9, *i.e.,* TACO, employs Information Retrieval tools to measure the conceptual dissimilarity between code elements of the source code.

Both the approaches using structural and other informations have good performances. Moreover, they are demonstrated to be complementary each other, *i.e.,* different approaches detect different code smell instances. However, despite the good results achieved by such tools, it is important to point out that a smell detection technique is actually useful only if it identifies code design problems that are recognized as such by software developers. In the context of this Chapter, we start comparing the usefulness of different types of information from the developers' perspective. In particular, we conducted a qualitative study–twelve developers of four open source systems–aimed at investigating to what extent the smells detected by HIST and by the single snapshot techniques compared in Chapter 7, *i.e.,* JDeodorant [25, 67] (for the *Feature Envy* smell), the *DECOR's* [20] detection rules (for the *Blob* smell), and the approach by Rao *et al.* [79] (for *Divergent Change* and *Shotgun Surgery*), reflect developers perception of poor design and implementation choices. Results of this study highlight that over 75% of the smell instances identified by HIST are considered as design/implementation problems by developers, that generally suggest refactoring actions to remove them.

8.2 Comparing the Developers' Perception of Historical and Structural Code Smells

The *goal* of the empirical study is to investigate (i) to what extent developers are able to perceive the code smells detected by HIST, *i.e.,* the historical-based approach presented in Chapter 7, and (ii) whether code smells detected using historical or structural analysis are closer to the developers' perception of design flaws. Indeed, despite having achieved good results in terms of detection capability, it is

also important to point out that a smell detection technique is actually useful only if it identifies code design problems that are recognized as *relevant problems* by developers. For this reason, we carried out a qualitative study having as objects the code smells detectable using both the types of information, summirized in Table 8.1 together with a short description.

In the study, we compared the code smell instances identified by HIST with the ones detected using other structural-based tools, such as JDeodorant [25, 67] for the *Feature Envy* smell, the *DECOR's* [20] detection rules for the *Blob* smell, and the approach by Rao *et al.* [79] for *Divergent Change* and *Shotgun Surgery*.

Table 8.1: Code smells detected by HIST

Code Smell	Brief Description
Divergent Change	A class is changed in different ways for different reasons
Shotgun Surgery	A change to the affected class (i.e., to one of its fields/methods) triggers many little changes to several other classes
Parallel Inheritance	Every time you make a subclass of one class, you also have to make a subclass of another
Blob	A class implementing several responsibilities, having a large number of attributes, operations, and dependencies with data classes
Feature Envy	A method is more interested in another class than the one it is actually in

The design of this second study was based on the results obtained in the study on the accuracy presented in Chapter 7. Specifically:

- For purely historical smells (*i.e., Divergent Change*, *Parallel Inheritance*, and *Shotgun Surgery*) we consider only instances that are identified by HIST. Indeed, the results discussed in Section 7 demonstrate low complementarity between HIST and the competitive techniques for detecting these smells, with HIST playing the major role.

- For the structural smells (*i.e., Blob* and *Feature Envy*) we consider instances identified (i) only by HIST (*onlyHIST* group), (ii) only by the competitive

technique (*onlyDECOR* for *Blobs* and *onlyJD* for *Feature Envy*), and (iii) by both techniques (*both* group). Indeed, the results achieved for these two smells show that historical and structural information can both be good alternatives for identifying smells. Thus, it is interesting to understand which of the above mentioned groups contains smells that are recognized as actual problems by developers.

8.2.1 Empirical Study Definition and Design

In the following, we report the design and planning of the study, by detailing the context selection, the research questions, the data collection process, as well as the analysis method.

Context Selection

A needed requirement for this study is, of course, software developers. In order to recruit participants, we sent invitations to active developers of ten of the twenty systems considered in our first study. In particular, we just considered systems exhibiting instances of at least three of the code smells investigated in this chapter. The active developers have been identified by analyzing the systems' commit history[1]. In total, we invited 109 developers receiving responses from twelve of them: two developers from `Apache Ant`, two from `Eclipse`, two from `Android SDK`, and six from `Android frameworks-base`. Note that, even if the number of respondents appears to be low (11% of response rate), we are inline with the suggested minimum response rate for the survey studies defined below 20% [167].

Research Questions

The study aims at addressing the following two research questions:

- **RQ1**: *Are the historical code smells identified by HIST recognized as design problems by developers?* This research question focuses its attention on the *Divergent Change*, *Parallel Inheritance*, and *Shotgun Surgery* smells. HIST is the

[1]We considered developers that performed at least one commit in the last two years.

first technique able to effectively identify instances of these smells. Thus, it is worthwhile to know if the instances of the smells it identifies really represent design problems for developers.

- **RQ2**: *Which detection technique aimed at identifying structural code smells better reflects developers' perception of design problems?* This research question aims at investigating how developers working on the four open-source systems perceive the presence of structural code smells identified by different detection techniques. In particular, we focus on smells identified by HIST only, by the techniques based on code analysis of a single snapshot only, and by both.

We answer both research questions through a survey questionnaire that participants filled-in online.

Survey Questionnaire Design

We designed a survey aimed at collecting developers' opinions needed to answer two of our research questions. Specifically, given the subject system S_i, the following process was performed:

1. ***Smell Instances Selection***. The smell instances to consider in our study were selected as follows:

   - For each **purely historical** smell c_j (*i.e.*, *Divergent Change*, *Parallel Inheritance*, and *Shotgun Surgery*) having at least one instance in S_i detected by HIST, we randomly selected one instance or took the only one available. Note that we refer to the "instance" as code component(s) affected by the smell. For example, it could be a single class affected by the *Divergent Change* smell, as well as a pair of classes affected by the *Parallel Inheritance* smell.
   - For each **structural** smell c_j (*i.e.*, *Blob* and *Feature Envy*) having at least one instance in S_i we randomly selected (i) one instance detected only by HIST (if any), (ii) one instance detected only by the DECOR or JDeodorant (if any), and (iii) one instance detected by both techniques (if any).

Table 8.2: Smell Instances Selected for each System.

System	DC	PI	SS	Blob			Feature Envy		
				onlyHIST	onlyDECOR	both	onlyHIST	onlyJD	both
Apache Ant	-	1	-	1	1	1	1	1	-
Eclipse	1	1	1	1	1	-	1	-	-
Android sdk	1	1	-	1	1	1	1	-	-
Apache frameworks-base	-	1	1	1	1	-	1	1	1
Overall	**2**	**4**	**2**	**4**	**4**	**2**	**4**	**2**	**1**

Note that this study excluded entities affected by more than one smell instance (*e.g.*, a method affected by both *Shotgun Surgery* and *Feature Envy*).

The smells selected on each system are summarized in Table 8.2. For sake of clarity, we abbreviated *Divergent Change*, *Parallel Inheritance*, and *Shotgun Surgery* in DC, PI, and SS, respectively. As it can be seen, we were not able to get the same number of instances for all the smells and for all the groups of structural smells. However, we were able to cover all smells and groups of smells (*i.e., onlyHIST*, *onlyDECOR/JD*, *both*) with at least one smell instance.

2. ***Defining Survey Questions***. For each selected smell instance, study participants had to look at the source code and answer the following questions:

   - In your opinion, does this code component[2] exhibit any design and/or implementation problem?
   - If YES, please explain what are, in your opinion, the problems affecting the code component.
   - If YES, please rate the severity of the design and/or implementation problem by assigning a score on the following five-points Likert scale [165]: 1 (very low), 2 (low), 3 (medium), 4 (high), 5 (very high).
   - In your opinion, does this class need to be refactored?
   - if YES, how would you refactor this class?

[2]Depending on the smell object of the question, a code component could be a method, a class, or a pair of classes.

On the one side, for questions related to **purely historical** smell instances detected by HIST, we also added hints on the change history of the code component (*i.e.,* the same information exploited by HIST to detect that smell instance). This was needed to provide participants with information related to the historical behavior of the involved code components. Indeed, it is impossible to spot a problem as a *Parallel Inheritance* without knowing the number of times the addition of a subclass to a class C_i also resulted in the addition of a subclass to a class C_j. On the other side, for **structural** smells, no metrics were shown for instances identified by HIST as well as by the competitive techniques.

The questionnaires included six tasks[3] for `Apache Ant`, `Eclipse JDT`, and `Android sdk`, and seven tasks for `Apache frameworks-base`.

Besides the above described survey, we also asked participants to fill-in a brief pre-questionnaire in order to assess their background. In particular, we asked:

- How many years of experience do you have in programming?
- How many years of experience do you have in industry?
- Rate your programming skills from 1=very low to 5=very high.

Note that all the questions in the survey, as well as the background questions prefacing the survey, were designed to make sure that the survey could be completed within approximately 60 minutes. This is why we limited (i) the number of tasks and (ii) the number of questions in the background section, since a higher number could have resulted in a higher dropout rate before even starting the main survey.

[3]By "task" we refer to the set of questions provided to a participant for each of the evaluated smell instances.

Data Collection

To automatically collect the answers, the survey and background questions were hosted on a Web application, *eSurveyPro*[4]. Developers were given 40 days to respond to the survey. Note that the Web application allowed developers to complete a questionnaire in multiple rounds, *e.g.,* to answer the first two questions in one session and finish the rest sometime later. At the end of the response period, we collected developers' answers in a spreadsheet in order to perform data analysis. As explained before, in the end we collected 12 complete questionnaires (two developers from `Apache Ant`, two from `Eclipse`, two from `Android SKD`, and six from `Android Frameworks-base`). Note that the developers of the four systems were invited to evaluate only the smells identified from the system that they were working on. Indeed, we are interested in gathering only data coming from original developers having sufficient knowledge of the analyzed source code components. Also, developers were not aware of the code smell types investigated in our study.

Analysis Method

To answer **RQ1** we computed, for each type of historical smell:

1. The percentage of cases where the smell has been *perceived* by the participants. By *perceived*, we mean cases where participants answered *yes* to the question: "In your opinion, does this code component exhibit any design or coding problem?"

2. The percentage of times the smell has been *identified* by the participants. The term *identified* indicates cases where besides perceiving the smell, participants were also able to identify the exact smell affecting the analyzed code components, by describing it when answering to the question "If yes, please explain what are, in your opinion, the problems affecting the code component". We consider a smell as *identified* only if the design problems described

[4] `http://www.esurveyspro.com` verified on September 2014.

by the participant are clearly traceable onto the definition of the smell affecting the code component. For example, given the following smell description for the *Feature Envy* smell: *"a method making too many calls to methods of another class to obtain data and/or functionality"*, examples of "correct" descriptions of the problem are *"the method is too coupled with the C_i class"*, or *"the method invokes too many methods of the C_i class"* where C_i is the class *envied* by the method. On the other hand, an answer like *"the method performs too many calls"* is not considered as sufficient to mark the smell as *identified*.

3. Descriptive statistics of answers provided by the participants to the question "please rate the severity of the coding problem". Note that for this point we just considered answers provided by developers that correctly *identified* the code smell.

4. The percentage of participants that answered yes to the question "does this class need to be refactored?". For participants answering "yes", we also report their responses to the question "how would you refactor this class?".

By performing this analysis for each historical code smell we should be able to verify if the instances of historical smells detected by HIST represent actual design problems for original developers.

As for **RQ2**, we perform the same exact analysis for each structural smell as described above for the historical smells. In addition, we compared the answers provided by participants for smell instances falling into three different categories (*i.e.*, *onlyHIST*, *onlyDECOR/onlyJD*, and *both*). Given the limited number of data points, this comparison is limited to descriptive statistics only, since we could not perform any statistical tests.

Replication Package

All the data used in our second study are publicly available [195]. Specifically, we provide: (i) the text of the e-mail sent to the developers; (ii) the raw data for the answers (anonymized) provided by the developers.

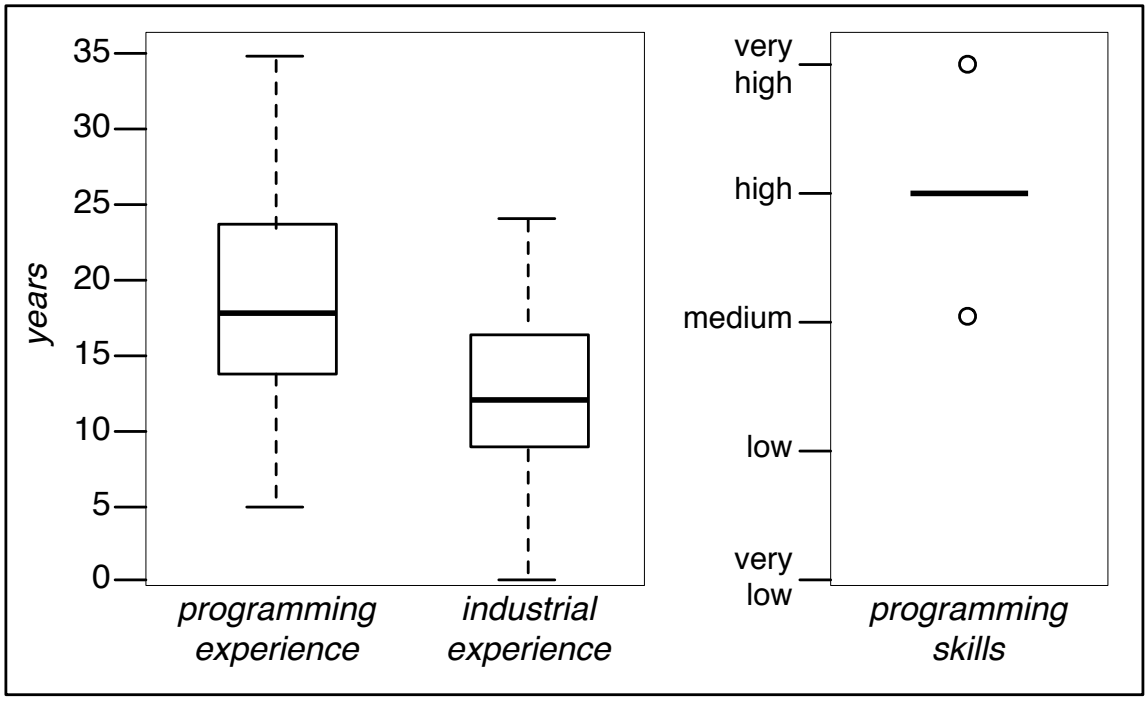


Figure 8.1: Experience of the involved developers.

8.2.2 Analysis of the Results

Before discussing the results of our two research questions, it is worthwhile to comment on the experience of the developers involved in our study. Fig. 8.1 reports the boxplots of the distribution of answers provided by developers to questions related to their experience in the background section. Twelve developers claimed a programming experience ranging between 5 to 35 years (mean=18.5, median=17.5), industrial experience ranging between 1 to 24 years (mean=12.7, median=12). Most of them rated their programming skills as *high*. Thus, all twelve participants had some sort of industrial experience and, most importantly, several years of programming experience.

Are the historical code smells identified by HIST recognized as design problems by developers? Figure 8.2 reports the percentage of developers that *correctly identified* the smell present in the code component. As explained in the design, we

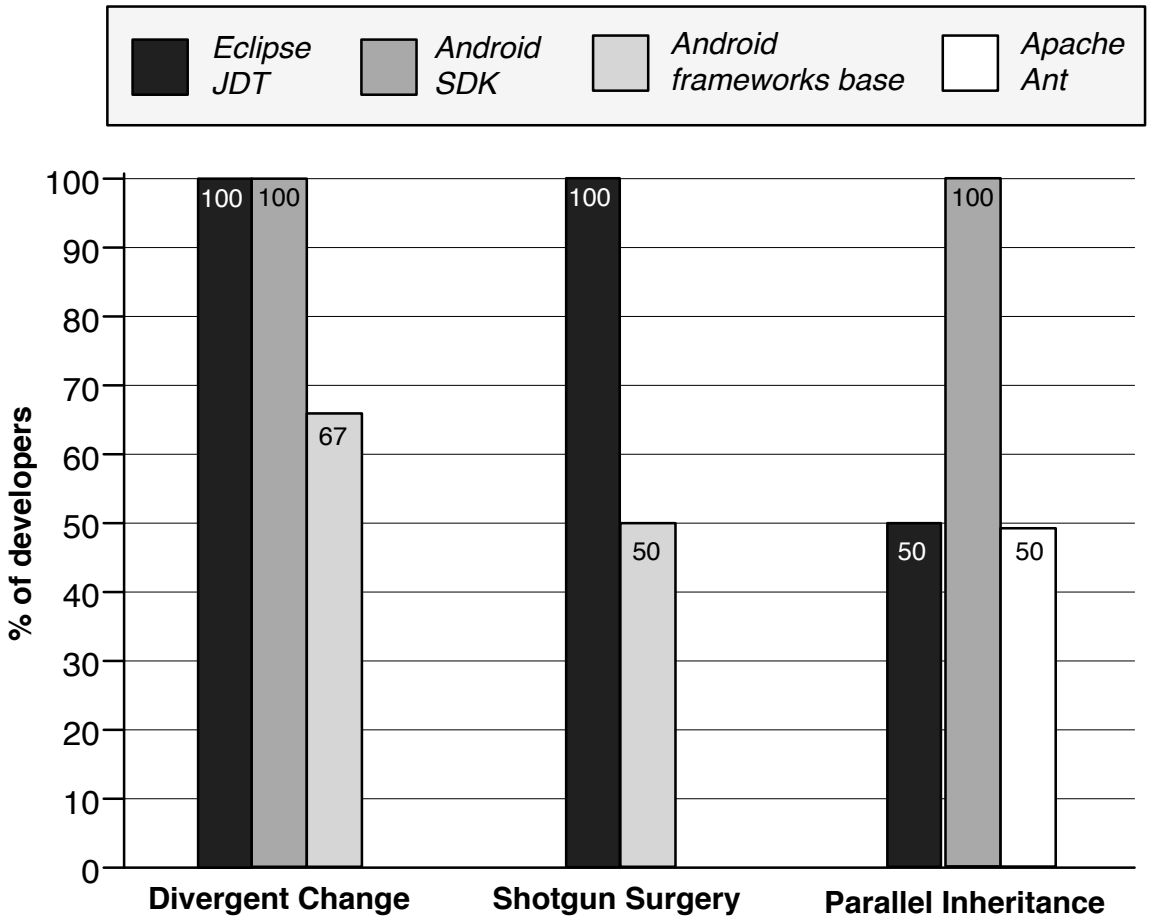


Figure 8.2: **RQ1**: percentage of *identified* smell instances.

computed both the percentage of developers that *perceived* and *identified* the smell[5]. However, in the context of this research question all developers who *perceived* the smell were also able to *identify* it.

In addition, Figure 8.3 reports the percentage of developers assigning each of the five levels of severity (going from *very low* to *very high*) to the identified design/implementation problems. Finally, Table 8.3 reports the percentage of developers that suggested refactoring operations for the identified smells. Their answers on how to refactor the smells are discussed in the text.

Starting from the *Divergent Change* instances identified by HIST, Figure 8.2 shows that developers generally recognized them as a design/implementation

[5]Note that the percentage of identified smells is a subset of the perceived one (see Section 8.2.1).

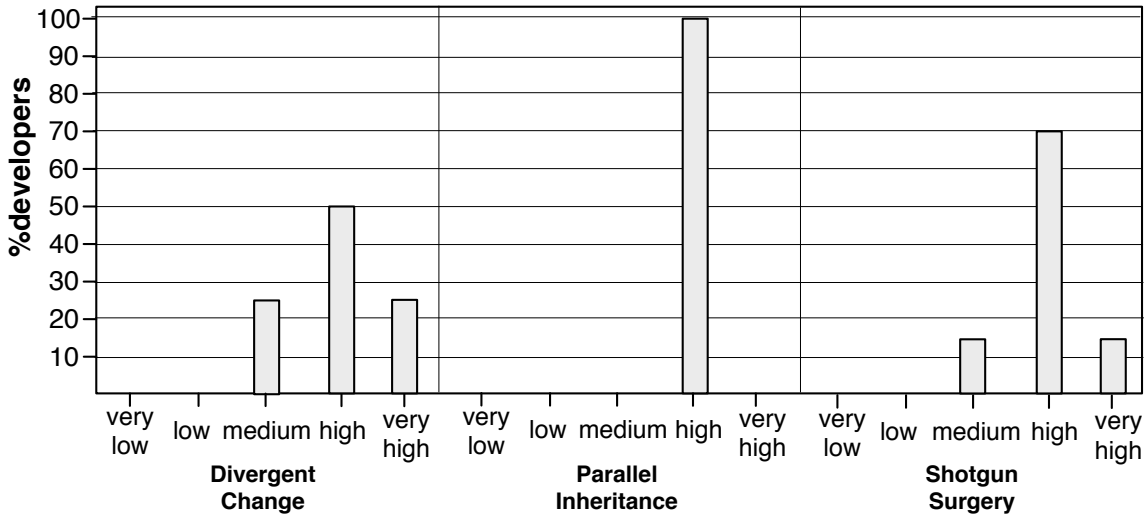


Figure 8.3: **RQ1**: severity assigned by developers to the *identified* instances of poorly historical smells detected by HIST.

Table 8.3: **RQ1**: percentage of developers in favor of refactoring the class among those correctly identifying the smells.

Code Smell	% in favor
Divergent Change	100%
Parallel Inheritance	100%
Shotgun Surgery	75%

problems. Indeed, the two `Eclipse JDT` developers, the two `Android sdk` developers, and four out of the six involved `Android frameworks-base` developers were able to perceive and identify the presence of a *Divergent Change* instance in the analyzed code components. Most of these developers pointed out low cohesion of the class as the root cause for the identified design problem. Low cohesion of classes is clearly a symptom of a *Divergent Change* smell. Indeed, classes having low cohesion tend to implement different responsibilities, that are likely to be changing divergently during time. Interesting is the refactoring suggested by one of the developers of `Android frameworks-base` recognizing this smell in the `PackageManagerService` class:

> *Make a new separate helper class for talking to the phone's file system.*

In other words, the developer is suggesting performing an Extract Class refactoring aimed at removing one responsibility from the `PackageManagerService`, and in particular the management of the phone file system. Concerning the severity of the problem as assessed by the developers identifying the smell, Figure 8.3 shows that 25% of them rate the severity as *medium*, 50% as *high*, and 25% as *very high*. Also, all of them agreed on the need to refactor the classes affected by *Divergent Change*.

As for the *Shotgun Surgery* smell, we have instances of this smell just in two out of the four subject systems (*i.e.*, `Eclipse JDT` and `Android frameworks-base`). The two involved `Eclipse JDT` developers recognized presence of this smell, explaining how *the high coupling of some of the methods contained in the* `MethodLocator` *class could represent a design/implementation problem*. Indeed, the basic ingredient for the appearance of a *Shotgun Surgery* smell is to have methods depending on several other classes.

Three of the `Android framework-base` developers (50%) identified the presence of a *Shotgun Surgery* instance in the `Handler` class as an implementation/design problem. One of them pointed out that:

> *Handler is tightly coupled to a few other classes: Message, MessageQueue and Looper. Each class has knowledge about members of the other classes. From a strict Object Oriented Programming perspective this is not optimal.*

However, the developer explained that from his point of view in this case the class affected by the smell should not be refactored, because:

> *At first glance the coupling looks like a problem, but these classes are best viewed as one unit. If you accept that perspective, the design problem just isn't there. There may also be performance benefits of accessing members in the other classes directly. For example, mLooper.mQueue instead of mLooper.getQueue(). It makes sense to trade design for performance for a class at the very core of the message loop.*

This example shows exactly what a smell is all about: it is a symptom in the code that may (or may not) indicate a design problem. Also, the example highlights the importance of this evaluation. Indeed, a smell detection tool should be able to point out smell instances representing an implementation/design problem that software developers are interested in refactoring. Note that the developer above is the only one who did not recognize the need to refactor `Handler` class. Concerning the severity of the identified *Shotgun Surgery* instances, 70% of developers assessed the severity as *high*, 15% to *very high*, and the remaining 15% to *medium*. Thus, the instances of *Shotgun Surgery* identified by HIST are mostly recognized as serious problems by the developers of these subject systems.

The *Parallel Inheritance* smell affects three of the subject systems (see Figure 8.2). This smell was the one among the least perceived (and identified) by developers. Still, one of the two involved developers of `Eclipse JDT` and `Apache Ant` systems as well as both the developers of `Android SDK` recognized its presence, talking about *problems in the design hierarchy*. All four developers recognizing the smell, assessed its severity as *high* and suggested to refactor it by moving responsibilities across the hierarchies. This could be done by applying move method refactoring as well as pull up/push down method/field refactorings.

Which detection technique aimed at identifying structural code smells better reflects developers' perception of design problems? Starting from the *Blob* smell, Figure 8.4 reports the percentage of developers who *perceived* (the striped columns) and *identified* (the filled columns) the *Blob* instances belonging to the *onlyHIST*, *onlyDECOR*, and *both* groups. Also, the left part of Figure 8.5 reports the percentage of developers assigning each of the five severity levels (going from *very low* to *very high*) to the identified *Blobs*. Finally, the top part of Table 8.4 reports the percentage of developers that suggested a refactoring for the identified *Blobs*.

We have instances of *Blobs* identified only by HIST on all four subject systems. Among the twelve involved developers, only one developer of `Android frameworks-base` did not recognize the evaluated *Blob* instance belonging to the *onlyHIST* group. The remaining eleven developers (92%) clearly described the

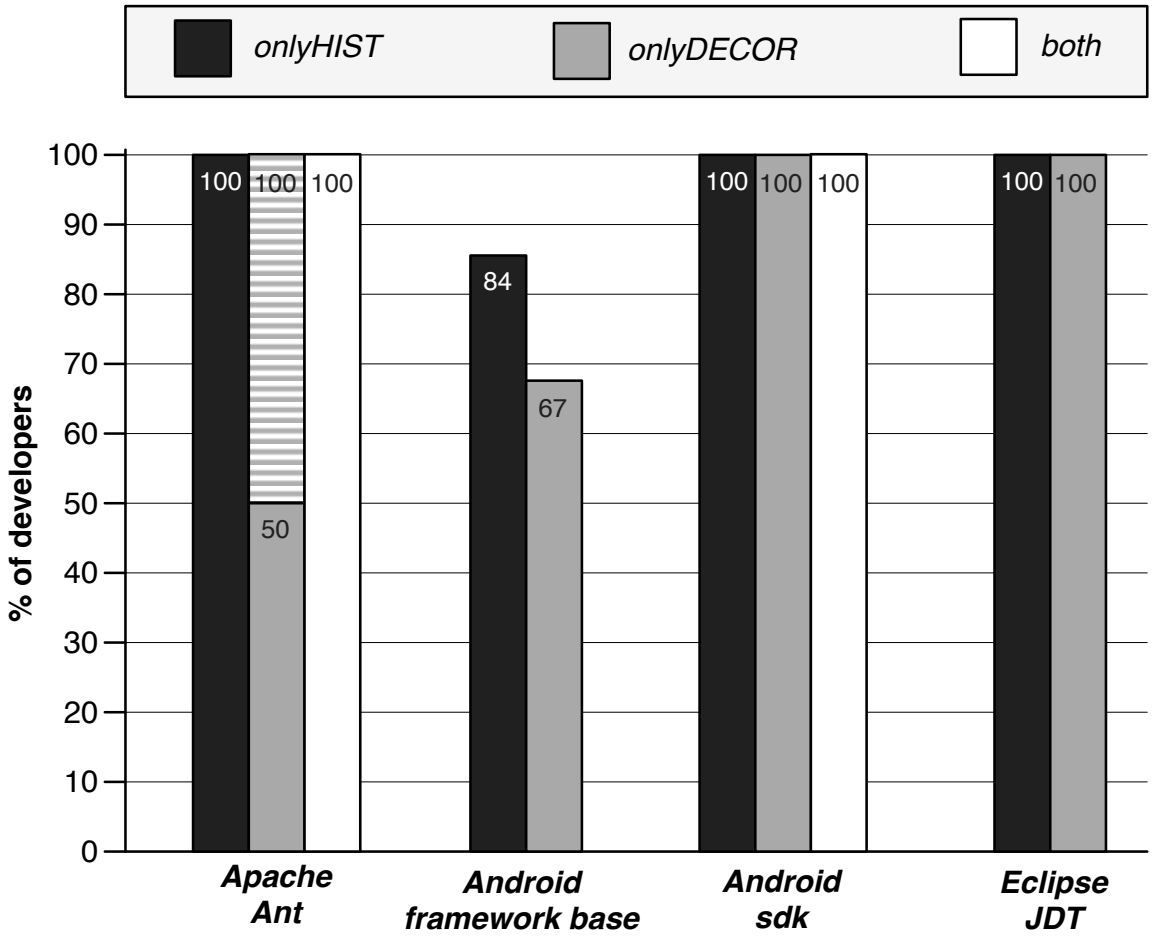


Figure 8.4: **RQ2**: percentage of *perceived* and *identified Blob* instances.

problem affecting the analyzed class. For example, an `Eclipse JDT` developer, referring to the analyzed class `SourceMapper`, wrote: *"this is a well known Blob in Eclipse"*; an `Android frameworks-base` developer explained, evaluating the class `WindowManagerService`: *"it is a very large and complex class"*. The eleven developers recognizing the *Blob* instances also evaluated the severity of the problem as *high* (18%) or *very high* (82%)—see left part of Figure 8.5—manifesting the willingness to refactor such classes in 100% of cases (see Table 8.4). Most of the developers suggested to perform an Extract Class refactoring to remove the smell (*e.g., "make the class easier to comprehend by splitting its responsibilities into different classes"*, from an *Android frameworks-base* developer). Thus, the *Blob* instances detected by HIST and missed by the competitive technique (*i.e.,* DECOR) have been

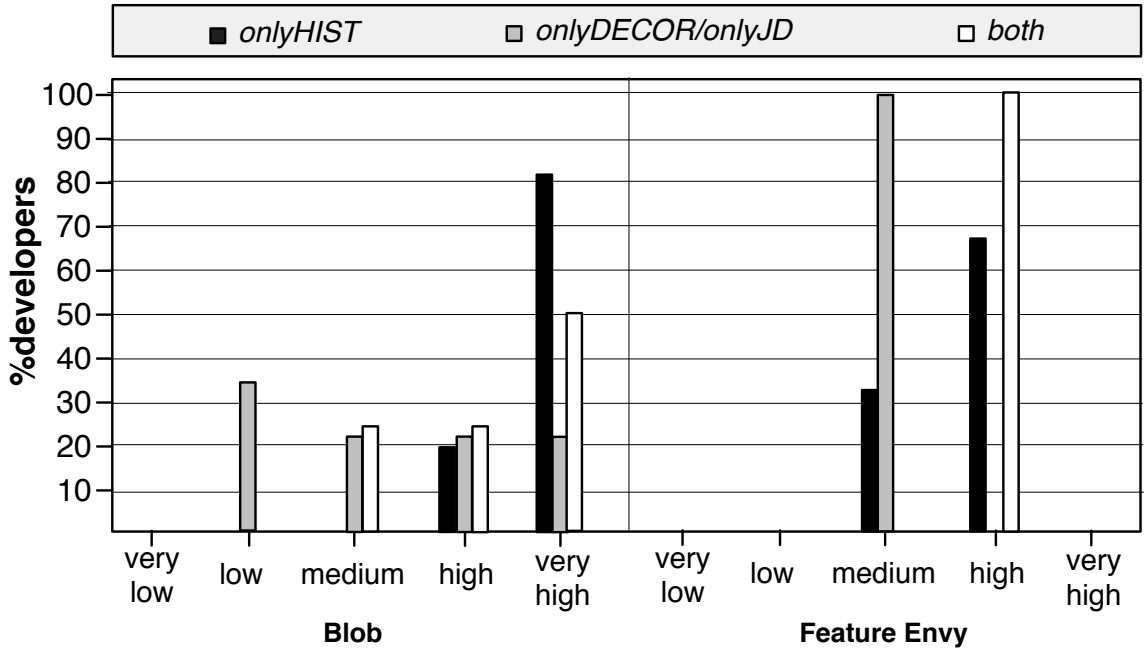


Figure 8.5: **RQ2**: severity assigned by developers to the *identified* instances of poorly historical smells detected by HIST (*onlyHIST*), by the competitive technique (*onlyDECOR* or *onlyJD*), and by *both*.

mostly recognized by developers as design/implementation problems. Also, the developers recognized the high severity of the issue caused by the presence of the smell, manifesting the willingness to refactor such classes.

As for the *Blob* instances detected by DECOR and missed by HIST, nine out of the twelve developers (75%) recognized them as design/implementation problems. In addition, one of the `Apache Ant` developers *perceived* the smell but failed to identify it[6] (see Figure 8.4). Concerning the severity assessed for the *Blob* instances identified in the *onlyDECOR* group, Figure 8.5 shows that 34% of developers selected a *low* severity, 22% *medium*, 22% *high*, and 22% *very high*. Also, 78% of developers recognized the need to refactor those *Blob* instances.

The third group of *Blob* instances to analyze is the one grouping together *Blobs* detected by both HIST and DECOR (*both* groups). We have instances of these *Blobs* only in `Apache Ant` and `Android SDK`. Interestingly, all developers recognized

[6]The developer described problems in a method manipulating jar files.

Table 8.4: **RQ2**: percentage of developers in favor of refactoring the class among those correctly identifying the smells.

Code Smell	Detected by	% in favor
Blob	*onlyHIST*	100%
	onlyDECOR	78%
	both	100%
Feature Envy	*onlyHIST*	100%
	onlyJD	100%
	both	100%

the *Blob* instances belonging to the *both* group, even if the severity assigned to them is lower than the severity assigned to the instances belonging to the *onlyHIST* group (see Figure 8.5). This result is quite surprising. Indeed, one would expect a very high severity for smells identified by both detection techniques. Still, the assessed severity is *medium* (25%), *high* (25%), or *very high* (50%). Moreover, in 100% of the cases developers agreed on the importance of refactoring the *Blob* instances belonging to the *both* group.

Summarizing, the *Blob* instances detected by both techniques are the ones that are mostly recognized by developers (100% of the developers), followed by the ones detected by HIST only (95%) and those detected by DECOR only (75%). The instances recognized as more severe problems are those identified by HIST only (82% *very high*), followed by those detected by both techniques (50% *very high*), and those detected by DECOR only (22% *very high*). Finally, all the developers agreed on refactoring the *Blob* instances detected by both techniques as well as those detected by HIST only, while 78% of developers agreed on refactoring the *onlyDECOR* instances.

Thus, when comparing HIST to DECOR, the *Blob* instances detected by DECOR only are (i) identified by fewer developers, (ii) evaluated with a much lower severity level, and (iii) recognized as less likely refactoring opportunities by developers.

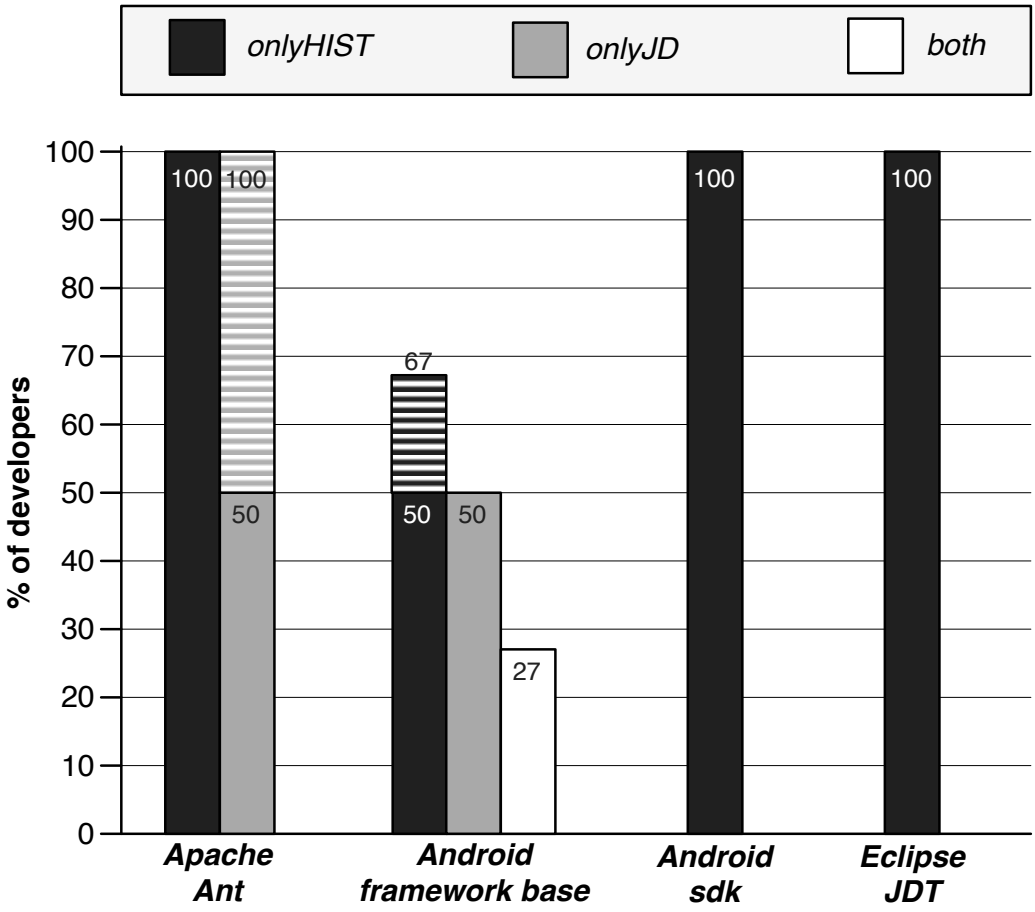


Figure 8.6: **RQ2**: percentage of *perceived* and *identified Feature Envy* instances.

Still, the fact that 75% of developers recognized the smells points out to the conclusion that complementing HIST with structural information (*e.g.,* DECOR) could be a worthwhile direction in order to identify currently missed *Blob* instances. This result confirms the results of our first study, further highlighting complementarity of the two techniques.

Turning the attention on the *Feature Envy* smell, Figure 8.6 shows the percentage of developers who *perceived* (the striped columns) and *identified* (the filled columns) the *Feature Envy* instances belonging to the *onlyHIST*, *onlyJD*, and *both* groups. As before, Figure 8.5 (right part) reports the severity assigned by developers to the identified smell instances, while Table 8.4 reports the percentage of developers that would like to refactoring the smell.

The *Feature Envy* instances falling in the *onlyHIST* group (black columns in Figure 8.6) have been recognized as design/implementation problems by nine out of twelve (75%) involved developers. In particular, all the developers of `Apache Ant`, `Android sdk`, and `Eclipse JDT` identified the smell instances, while only three of the six `Android framework base` developers recognized the problem. Developers recognizing the problem generally described the issue explaining that the analyzed method has high coupling with another class (*i.e.*, the envied class). For example, while analyzing the method `generateCode` of the class named `AND_AND_Expression` (`Eclipse JDT` project), one of the developers explained that *"generateCode is a very complex method and it is highly coupled with the CodeStream class"*. `CodeStream` is exactly the class identified by HIST as the envied class for the `generateCode` method.

Concerning the severity assigned to the smell by the nine developers identifying it, 67% rated it as *high*, while 33% as *medium* (see Figure 8.5). Moreover, all nine developers suggested to refactor this smell (see Table 8.4) proposing a *Move Method* toward the envied class, or an *Extract Method* followed by a *Move Method*.

As for the *Feature Envy* instances identified by JDeodorant only, we have instances of them on the `Apache Ant` and the `Android framework base` systems. On `Apache Ant` both developers perceived a problem in the analyzed *Feature Envy* instance (*i.e.*, the `run` method from the `ClearCase` class), but only one correctly identified the smell. On the `Android framework base`, among the six involved developers three identified a *Feature Envy* in the method under analysis (*i.e.*, `executeLoad` from the `FrameLoader` class). Thus, four out of the eight evaluators (50%) identified the *Feature Envy* instances in the *onlyJD* group. All of them assessed the severity of the spotted instances of the smell as *medium*, manifesting some readiness to refactor them.

Finally, the only instance falling in the *both* group belongs to the `Android SDK` system. This instance has been identified by both involved developers, that assessed its severity as *high*, and suggested a *Move Method* refactoring to solve the problem. This confirms, in part, what we observed for the *Blob* smell: when both HIST and the techniques based on a single snapshot analysis detect a code smell,

all involved developers identify the smell and suggest appropriate refactoring operation.

Summarizing, the *Feature Envy* instances detected by both techniques are the most recognized by developers (100% of developers), followed by the ones detected by HIST only (75%) and those detected by JDeodorant only (38%). Also, the instances recognized as more severe problems are those detected by both techniques (100% *high*), followed by those detected by HIST only (67% *high*), and those detected by JDeodorant only (100% *medium*). Despite these differences, all the developers identifying the *Feature Envy* instances falling in the three different groups (i.e., *onlyHIST*, *onlyJD*, and *both*) suggested to refactor them.

8.3 Threats to Validity

This section discusses the threats that could affect the validity of the study.

Construct Validity. Threats to *construct validity* are mainly related to how we measured the developers' perception of smells. As explained in Section 8.2.1, we asked developers to tell us whether they perceived a problem in the code shown to them. In addition, we asked them to explain what kind of problem they perceived to understand whether or not they actually identify the smell affecting the code component as the design and/or implementation problem. Finally, for the severity we used a Likert scale [165] that permits the comparison of responses from multiple respondents. We are aware that questionnaires could only reflect a subjective perception of the problem, and might not fully capture the extent to which the smell instances identified by HIST and by the competitive techniques are actually perceived by developers.

Internal Validity. A factor that could have affected the results of the study is the response rate: while appearing not very high (11%), it is inline what it is normally expected in survey studies (*i.e.,* below 20% [167]). Note also that we just targeted for this study original developers of the four open source systems, without taking into account the possibility of involving students or people with no experience on

the object systems. Still, we cannot ensure that the involved developers had a good knowledge of the specific code components used in our surveys. An alternative design would have been to invite only developers actually involved in the development of the specific code components evaluated in our survey. However, (i) the different code components present in our survey are evolved and maintained by different developers, and (ii) this would have resulted in a much lower number of developers invited, having as a consequence a very likely drop in the number of participants in our study.

Also, we tried to keep the questionnaire as short as possible to have more developers answering our survey. For instance, we did not include any questions on non-smelly code entities as sanity check in our survey. Thus, we cannot exclude that participants always indicated that the analyzed code components contained a design/implementation problem and the problem was a serious one. However, this holds for the smell instances identified by HIST as well as for those identified by the competitive techniques.

It must be clear that even if developers recognized most of the code smell instances identified by HIST and declared that they wanted to refactor them, this does not always mean that it is possible to take proper refactoring actions aimed at removing those smells. Indeed, some systems—*e.g.*, `Eclipse JDT`—contain classes that naturally tend to become smelly. For example, parsers (largely present in the `Eclipse JDT`) are often affected by the *Blob* code smell [27], and are difficult to remove without taking important (and expensive) refactoring actions.

External Validity. Threats to *external validity* concern the generalization of the results. In this category, they can be related to the set of chosen objects and to the pool of the participants to the study. Concerning the chosen objects, we are aware that our study is based on smell instances detected in four Java systems only, and that further studies are needed to confirm our results. In this study we had to constrain our analysis to a limited set of smell instances, because the task to be performed by each respondent had to be reasonably small (to ensure a decent response rate).

8.4 Conclusion

In this Chapter, we conducted a qualitative study aimed at understanding to what extent the code smells identificable by HIST are considered as actual design problems by the developers. We also evaluate what is the difference in terms of perception between historical and structural code smells. To this aim, we involved twelve original developers of four open source systems. The results achieved indicated that over 75% of smell instances identified by HIST are also recognized by developers as actual design/implementation problems. In addition, this study showed that smell instances identified by both HIST and the single-snapshot techniques are the ones that perfectly match developers' perception of design/implementation problems. This result highlights once again the need of having hybrid smell detection approaches able to efficiently combine static code analysis with analysis of co-changes. Finally, we would like to perform a deeper investigation into the characteristics causing a smell instance to represent/not represent a problem for developers.

Chapter 9

A Textual-based Approach for Code Smell Detection

9.1 Introduction

Continuous change requests, strict and close deadlines, the need to preserve the quality of source code to ease maintenance are just some of the challenges that developers must face every day. In such a scenario, finding the solution that provides the maximum gain from each point of view is quite impossible. Very often, due to time constraints or absence of software design documentation, developers decide to set aside good programming guidelines and implement a new change request in the most straightforward way. As a consequence, the original design tends to be eroded, leading to the introduction of *technical debts* [5]. One of the main causes for technical debts are represented by "*bad code smells*" (a.k.a., "code smells" or simply "smells") [8], namely poor design or implementation choices applied by programmers during the development of a software system.

Over the last decade, researchers investigated several aspects related to the presence of code smells, demonstrating (i) their relevance from the developers' perspective [50, 134], (ii) their longevity [14, 13, 15, 12, 107], and (iii) their impact on non-functional properties of source code, such as program comprehension [18], change- and fault-proneness [16, 17], and, more in general, on maintainability

[9, 118, 117, 2]. For these reasons the research community devoted a lot of effort to define methods to detect code smells in source code and, whenever possible, trigger refactoring operations [20, 25, 21, 23, 26, 22, 24]. These tools generally apply constraint-based detection rules defined on some source code metrics, *i.e.,* the majority of existing approaches try to detect code smells through the analysis of structural properties of code components (*e.g.,* methods).

Analyzing the catalogue of smells defined in the literature, it is easy to identify a specific family of smells that are represented by source code components with promiscuous responsibilities. For example, a *Blob* is a giant class that centralizes the behavior of a portion of the system and has a lot of different responsibilities, while a *Feature Envy* refers to a method more related to a different class with respect the one it is actually in. Even if these smells, such as Blob, are generally identified by considering structural properties of the code (see for instance [20]), there is still room for improving their detection by exploring other sources of information. Indeed, Palomba *et al.* [52] recently proposed the use of historical information for detecting several bad smells, including Blob. At the same time, components with promiscuous responsibilities may be identified also considering the textual coherence of the source code vocabulary (*i.e.,* terms extracted from comments and identifiers).

To explore this conjecture, we proposed **TACO** (**T**extual **A**nalysis for **C**ode smell detecti**O**n), a code smell detector purely based on Information Retrieval (IR) methods. Specifically, we instantiated TACO for detecting five code smells, *i.e., Long Method*, *Feature Envy*, *Blob*, *Promiscuous Package* and *Misplaced Class*. We empirically evaluated the performances of TACO on 20 open source software systems, finding that our detector achieves good precision and recall for all the considered smells. Moreover, when compared with state-of-the-art techniques solely based on structural analysis, *i.e.,* DECOR [20], JDeodorant [25], and the approaches proposed in [196] and [76], we experienced that most of the times TACO outperforms these existing approaches.

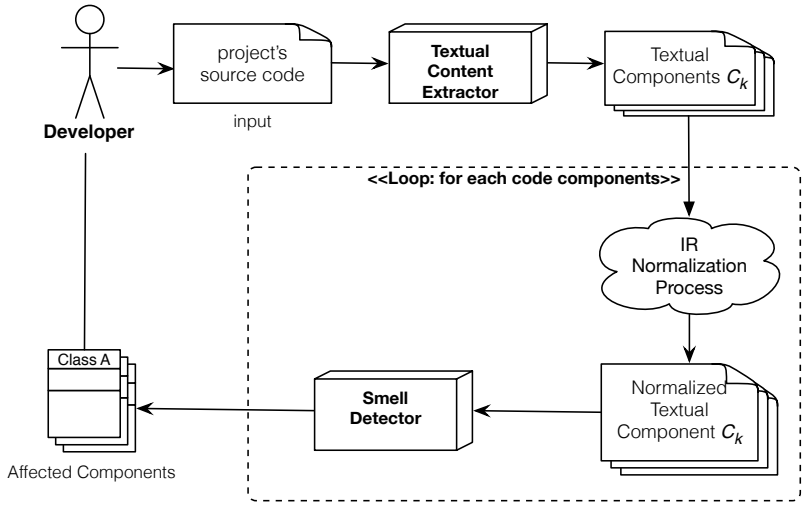


Figure 9.1: TACO: The proposed code smell detection process.

9.2 Detecting Code Smells Using Information Retrieval Techniques

Figure 9.1 depicts the main steps used by TACO in order to compute the probability of a code component being affected by a smell, which are (i) *Textual Content Extractor*, (ii) *IR Normalization Process*, and (iii) *Smell Detector*.

Textual Content Extractor. Starting from the set of code artifacts composing the software project under analysis, the first step is responsible for the extraction of the textual content characterizing each code component by selecting only the textual elements actually needed for the textual analysis process, *i.e.*, source code identifiers and comments.

IR Normalization Process. Identifiers and comments of each component are firstly normalized by using a typical Information Retrieval (IR) normalization process. Thus, the terms contained in the source code are transformed by applying the following steps [55]: (i) separating composed identifiers using the camel case split-

ting which splits words based on underscores, capital letters and numerical digits; (ii) reducing to lower case letters of extracted words; (iii) removing special characters, programming keywords and common English stop words; (iv) stemming words to their original roots via Porter's stemmer [197]. Finally, the normalized words are weighted using the *term frequency - inverse document frequency* (*tf-idf*) schema [55], which reduces the relevance of too generic words that are contained in most source components.

Smell Detector. The normalized textual content of each code component is then individually analyzed by the *Smell Detector*, which applies different heuristics to identify target smells. The detector relies on Latent Semantic Indexing (LSI) [198], an extension of the Vector Space Model (VSM) [55], which models code components as vectors of terms occurring in a given software system. LSI uses Singular Value Decomposition (SVD) [199] to cluster code components according to the relationships among words and among code components (co-occurrences). Then, the original vectors (code components) are projected into a reduced k space of concepts to limit the effect of textual noise. For the choice of size of the reduced space (k) we used the heuristic proposed by Kuhn *et al.* [200] that provided good results in many software engineering applications, *i.e.*, $k = (m \times n)^{0.2}$ where m denotes the vocabulary size and n denotes the number of documents (code components in our case). Finally, the textual similarity among software components is measured as the cosine of the angle between the corresponding vectors. The similarity values are then combined in different ways, according to the type of smell we are interested in, to obtain a probability that a code component is actually smelly. For detection purpose, we convert such a probability in a truth value in the set {`true, false`} to denote whether a given code component is affect or not by a specific smell. In the context of this chapter, we instantiated TACO for detecting five code smells, namely (i) *Long Method*, (ii) *Feature Envy*, (iii) *Blob*, (iv) *Promiscuous Package*, and (v) *Misplaced Class*. We have instantiated TACO on these code smells in order to demonstrate how textual analysis can be useful for detecting smells at different levels of granularity (*i.e.,* method-level, class-level,

and package-level). Moreover, all the selected smells violate, in different ways, the OOP single responsibility principle [201, 202]. For instance, a Blob class implements more than one responsibility, while a Feature Envy is a method which has a different responsibility with respect to the one implemented in the class it is actually in. This peculiar characteristic makes them particularly suitable for a textual-based technique, since *the higher the number of the responsibilities implemented in a code component, the higher the probability that such a component contains heterogeneous identifiers and/or comments*. In the following subsections, we detail how the general process depicted in Figure 9.1 has been applied for detecting the smells described above.

9.2.1 Computing Code Smell Probability

Long Method. This smell arises when a method implements a main functionality, together with auxiliary functions that should be managed in different methods. The refactoring associated with such a smell is clearly the *Extract Method*[1], which allows the identification of portions of the method that should be treated separately, with the aim to create new methods for managing them [8]. It is worth noting that the definition of the smell strongly differs from its name, since this smell is only partially related to the size of a method. Rather, it is related to how much responsibilities a method manages, *i.e.*, whether a method violates the single responsibility principle.

Textual Diagnosis. Given the definition above, our conjecture is that a method is affected by this smell *when it is composed of sets of statements semantically distant to each other*. In order to detect the different sets of statements composing the method, we re-implemented *SEGMENT*, the approach proposed by Wang *et al.* [203], which uses both structural analysis and naming information to automatically segment a method into a set of "*consecutive statements that logically implement a high level action*" [203]. Once we identified the sets of statements (*i.e.*, segments) composing

[1]More details about refactoring operations defined in literature can be found in the refactoring catalog available at http://refactoring.com/catalog/

the method, we considered each of them as a single document. Then, for each pair of documents, we apply LSI [204] and the cosine similarity to have a similarity value. More formally, let M be the method under analysis, let $S = \{s_1, \ldots, s_n\}$ be the set of segments in M, we compute the textual cohesion of the method M as the average similarity between all pairs of its segments:

$$\text{MethodCohesion}(M) = \underset{i \neq j}{\text{mean}} \; sim(s_i, s_j) \tag{9.1}$$

where n is the number of code segments in M, and $sim(s_i, s_j)$ denotes the cosine similarity between two segments s_i and s_j in M. Starting from our definition of textual cohesion of M, we compute the probability that M is affected by *Long Method* using the following formula:

$$P_{LM}(M) = 1 - \text{MethodCohesion}(M) \tag{9.2}$$

It is worth noting that $P_{LM}(M)$ ranges in $[0; 1]$. The higher its value, the higher the probability that method M represents a Long Method instance.

Feature Envy. According to Fowler's definition [8], this smell occurs when "*a method is more interested in another class than the one it is actually in*". Thus, a method affected by Feature Envy is not correctly placed, since it exhibits high coupling with a class different than the one where it is located in. To remove this smell, a *Move Method* refactoring aimed at moving it to the more suitable class is needed.

Textual Diagnosis. When computing the probability that a method is affected by such a smell, we conjecture that a *method more interested in another class is characterized by a higher similarity with the concepts implemented in the envied class, with respect to the concepts implemented in the class it is actually in*. Let M be the method under analysis belonging to the class C_O, and let $C_{related} = \{C_1, \ldots, C_n\}$ be the set of classes in the system sharing at least one term with M. First, we derive the class ($C_{closest}$) having the highest textual similarity with M as follows:

$$C_{closest} = \underset{C_i \in C_{related}}{\arg\max} \; sim(M, C_i) \tag{9.3}$$

Then, if $C_{closest}$ is not the class where M is actually placed in (*i.e.*, $C_{closest} \neq C_O$),

then, M should be moved to the class $C_{closest}$. Therefore, we compute the probability for M to be a *Feature Envy* instance as:

$$P_{FE}(M) = sim(M, C_{closest}) - sim(M, C_O) \tag{9.4}$$

The formula above is equal to zero when $C_{closest} = C_O$, *i.e.*, the method M is correctly placed. Otherwise, if $C_{closest} \neq C_O$, the probability is equal to the difference between the textual similarities of M and the two classes $C_{closest}$ and C_O.

Blob. These classes are usually characterized by a huge size, a large number of attributes and methods and a high number of dependencies with data classes [8]. This smell involves low cohesive classes that are responsible for the management of different functionalities. The *Extract Class* refactoring is the more suitable operation that can be applied for removing this smell type, since it allows to split the original class by creating new classes, re-distributing its responsibilities.

Textual Diagnosis. Our conjecture is that Blob classes *are characterized by a semantic scattering of contents*. More formally, let C be the class under analysis, let $M = \{M_1, \ldots, M_n\}$ be the set of methods in C, we compute the textual cohesion of the class C as defined by Marcus and Poshyvanyk [45]:

$$\text{ClassCohesion}(C) = \underset{i \neq j}{\text{mean}}\ sim(M_i, M_j) \tag{9.5}$$

where n is the number of methods in C, and $sim(M_i, M_j)$ denotes the cosine similarity between two methods M_i and M_j in C. Therefore, we compute the probability that C is affected by *Blob* using the following formula:

$$P_B(C) = 1 - \text{ClassCohesion}(C) \tag{9.6}$$

Also in this case, $P_B(C)$ ranges in $[0; 1]$.

Promiscuous Package. A package can be considered as promiscuous if it contains classes implementing too many features, making it too hard to understand and maintain [8]. As for Blob, this smell arises when the package has low cohesion, since it manages different responsibilities. In this case, to refactor the smell an *Extract Package* operation is needed for split the package in more sub-packages, re-organizing the responsibilities of the original promiscuous package.

Textual Diagnosis. We conjecture that packages affected by this smell *are characterized by subset of classes semantically distant from the other classes of the package*. Formally, let P be the package under analysis, and let $C = \{C_1, \ldots, C_n\}$ be the set of classes in P, the textual cohesion of the package P is defined as done by Poshyvanyk *et al.* [182]:

$$\text{PackageCohesion}(P) = \underset{i \neq j}{\text{mean}} \; sim(C_i, C_j) \tag{9.7}$$

where n is the number of classes in P, and $sim(C_i, C_j)$ is the cosine similarity between two classes C_i and C_j in P. Given such definition, we define the probability that P is a *Promiscuous Package* using the formula below:

$$P_{PP}(P) = 1 - \text{PackageCohesion}(P) \tag{9.8}$$

$P_{PP}(P)$ assumes values in the range $[0; 1]$.

Misplaced Class. A *Misplaced Class* smell suggests a class that is in a package that contains other classes not related to it [8]. The obvious way to remove such a smell is to apply a *Move Class* refactoring able to place the class in a more related package.

Textual Diagnosis. Our conjecture is that a class affected by this smell is *semantically more related to a different package with respect to the package it is actually in*. Let C be the class under analysis, contained in the package P_O, and let $P_{related} = \{P_1, \ldots, P_n\}$ be the set of packages that share at least one term with C. We firstly retrieve the package with the highest textual similarity with C, using the following formula:

$$P_{closest} = \underset{P_i \in P_{related}}{\arg\max} \; sim(C, P_i) \tag{9.9}$$

Then, if $P_{closest}$ is different from the package where the class C is actually placed in (*i.e.*, $P_{closest} \neq P_O$), then C should be moved to the package $P_{closest}$. More formally, we compute the probability C is affected by a *Misplaced Class* as:

$$P_{MC}(C) = sim(C, P_{closest}) - sim(C, P_O) \tag{9.10}$$

As in the case of *Feature Envy*, the value is equal to zero if $P_{closest} = P_O$. Otherwise, if $P_{closest} \neq P_O$, the probability is equal to the difference between the textual similarities of C and the two packages $P_{closest}$ and P_O.

9.2.2 Applying TACO to Code Smell Detection

TACO assigns a smelliness probability to each code component according to the textual diagnosis metrics reported above. In the context of smell detection, we need to convert such probabilities in a truth value in the set {`true, false`}. Thus, we need to discriminate when a probability indicates the presence of a given smell with respect to the cases where such probability is not enough for considering a code component affected by a smell. After different experiments aimed at identifying the optimal cut-off point, we found that the best results are obtained when using as threshold the median of the non-null values of the probability distribution of the system under analysis. Interested readers can find the results of such a calibration analysis in our online appendix [205].

9.3 The Accuracy of TACO

In this section we discuss the empirical study we conducted in order to evauate the ability of the proposed code smell detector in identifying code components affected by design flaws.

9.3.1 Empirical Study Definition and Design

The *goal* of the study is to evaluate TACO, with the *purpose* of investigating its effectiveness during the detection of code smells in software systems. The *quality focus* is on the detection accuracy and completeness when compared to the approaches based purely on structural analysis. The *perspective* is of researchers, who want to evaluate the effectiveness of textual analysis for detecting code smells for building better recommenders for developers.

Table 9.1: Characteristics of the Software Projects in Our Dataset

System	#Releases	#Commits	Classes	Methods	KLOCs
ArgoUML	16	19,961	777-1,415	6,618-10,450	147-249
Apache Ant	22	13,054	83-813	769-8,540	20-204
aTunes	31	6,276	141-655	1,175-5,109	20-106
Apache Cassandra	13	20,026	305-586	1,857-5,730	70-111
Eclipse Core	29	21,874	744-1,181	9,006-18,234	167-441
FreeMind	16	722	25-509	341-4,499	4-103
HSQLDB	17	5,545	54-444	876-8,808	26-260
Apache Hive	8	8,106	407-1,115	3,725-9,572	64-204
Apache Ivy	11	601	278-349	2,816-3,775	43-58
Apache Log4j	30	2,644	309-349	188-3,775	58-59
Apache Lucene	6	24,387	1,762-2,246	13,487-17,021	333-466
JEdit	29	24,340	228-520	1,073-5,411	39-166
JHotDraw	16	1,121	159-679	1,473-6,687	18-135
JVLT	15	623	164-221	1,358-1,714	18-29
Apache Karaf	5	5,384	247-470	1,371-2,678	30-56
Apache Nutch	7	2,126	183-259	1,131-1,937	33-51
Apache Pig	8	2,857	258-922	1,755-7,619	34-184
Apache Qpid	5	14,099	966-922	9,048-9,777	89-193
Apache Struts	7	4,297	619-1,002	4,059-7,506	69-152
Apache Xerces	16	5,471	162-736	1,790-7,342	62-201
Overall	**301**	**183,514**	**25-2,246**	**188-17,021**	**4-466**

The *context* of the study consists of twenty open source software projects. Table 9.1 reports the characteristics of the analyzed systems[2], namely the number of public releases, and their size in terms of number of commits, classes, methods, and KLOC. Among the analyzed projects, we have twelve projects belonging to the Apache ecosystem[3], and eight open source projects from elsewhere. Note that our choice of the subject systems is not random, but instigated by our aim to analyze projects belonging to different ecosystems, having different size and scope.

In this study, we investigate the following research questions:

- **RQ1:** *What is the accuracy of TACO in detecting code smells?*

[2]The list of repositories is available in our online appendix [205]

[3]`http://www.apache.org/` verified on November 2016

- **RQ2:** *How does TACO perform when compared with state-of-the-art techniques purely based on structural analysis?*
- **RQ3:** *To what extent is TACO complementary with respect to the structural-based code smell detectors?*

To answer **RQ1** we run TACO on the selected software projects. To evaluate its accuracy, we needed an oracle reporting the actual code smells contained in the considered systems. For all of the code smells considered in this chapter, an annotated set of such smells is publicly available in literature [60].

Once obtained the set of smells detected by TACO on each software project, we evaluated its performances by using two widely adopted Information Retrieval (IR) metrics, *i.e.,* precision and recall [55]:

$$precision = \frac{|TP|}{|TP \cup FP|}\% \qquad recall = \frac{|TP|}{|TP \cup FN|\%} \tag{9.11}$$

where TP and FP represent the set of true and false positive smells detected by TACO, respectively, while FN (false negatives) represents the set of smell instances in the oracle missed by TACO. To have an aggregate indicator of precision and recall, we also report the F-measure, defined as the harmonic mean of precision and recall:

$$F\text{-}measure = 2 * \frac{precision * recall}{precision + recall}\% \tag{9.12}$$

To answer **RQ2**, we run code smell detection techniques purely based on structural analysis on the same software projects on which we run TACO. For *Long Method* and *Blob* smells we compared TACO with DECOR, the structural detection approach proposed by Moha *et al.* [20]. Since a tool implementing the approach is not publicly available, we re-implemented the detection rules defined by DECOR, which are available online[4]. For the *Feature Envy* smell, we considered JDeodorant

[4]http://www.ptidej.net/research/designsmells/grammar/

as the alternative approach [25]. JDeodorant is available as open source Eclipse plug-in[5]. The technique behind JDeodorant analyzes attributes and method calls of each method in the system under analysis with the aim to form a set of candidate target classes where the method should be moved. As for *Promiscuous Package*, we compared TACO with the clustering-based algorithm proposed by Girvan *et al.* [196], where classes are grouped using the dependencies among them. Finally, for *Misplaced Class*, we used the approach proposed by Atkinson and King [76], which traverse the abstract syntax tree of a class in order to determine, for each feature, the set of classes referencing them. In this case, a class is affected by *Misplaced Class* if it has more dependencies with a different package with respect to the one it is actually in.

Even if in literature several other approaches have been defined for smell detection, our choice of the alternative techniques has been guided by (i) the availability of a tool (*e.g.*, JDeodorant), or (ii) the simplicity of a re-implementation, in order to avoid possible errors due to a wrong implementation. To compare the performance achieved by TACO with those of the alternative structural detection techniques, we used the same set of accuracy metrics used for measuring TACO's results, *i.e.*, recall, precision, and F-measures.

Finally, to answer **RQ3**, we compared the sets of smell instances correctly detected by TACO and by the alternative approaches by computing the following overlap metrics:

$$correct_{m_i \cap m_j} = \frac{|correct_{m_i} \cap correct_{m_j}|}{|correct_{m_i} \cup correct_{m_j}|}\% \tag{9.13}$$

$$correct_{m_i \setminus m_j} = \frac{|correct_{m_i} \setminus correct_{m_j}|}{|correct_{m_i} \cup correct_{m_j}|}\% \tag{9.14}$$

where $correct_{m_i}$ represents the set of correct code smells detected by the approach m_i, $correct_{m_i \cap m_j}$ measures the overlap between the set of true code smells detected

[5]http://www.jdeodorant.com/

Table 9.2: Long Method - TACO accuracy compared to DECOR.

	#Actual	TACO						DECOR					
Project	Smells	Det.	TP	FP	Prec.	Rec.	F-M	Det.	TP	FP	Prec.	Rec.	F-M
ArgoUML	49	60	47	13	78%	96%	86%	69	34	35	49%	69%	58%
Apache Ant	52	73	50	23	68%	96%	80%	46	29	17	63%	56%	59%
aTunes	11	12	9	3	75%	81%	78%	16	2	14	13%	18%	15%
Apache Cassandra	10	11	8	3	73%	80%	76%	12	6	6	50%	60%	55%
Eclipse Core	89	93	69	24	74%	78%	76%	304	73	231	24%	82%	37%
FreeMind	12	13	10	3	77%	83%	80%	12	8	4	67%	67%	67%
HSQLDB	75	92	64	28	70%	85%	77%	97	52	45	54%	69%	60%
Apache Hive	53	64	45	19	70%	85%	77%	75	35	40	47%	66%	55%
Apache Ivy	18	21	16	5	76%	89%	82%	12	9	3	75%	50%	60%
Apache Log4j	4	4	2	2	50%	50%	50%	5	2	3	40%	50%	44%
Apache Lucene	82	101	66	35	65%	80%	72%	172	58	114	34%	71%	46%
JEdit	12	16	10	6	63%	83%	71%	19	6	13	32%	50%	39%
JHotDraw	13	18	11	7	61%	85%	71%	36	5	31	14%	38%	20%
JVLT	7	8	5	3	63%	71%	67%	5	3	2	60%	43%	50%
Apache Karaf	21	24	19	5	79%	90%	84%	14	10	4	71%	48%	57%
Apache Nutch	17	21	15	6	71%	88%	79%	12	7	5	58%	41%	48%
Apache Pig	33	38	28	10	74%	85%	79%	111	21	90	19%	64%	29%
Apache Qpid	39	46	32	14	70%	82%	75%	53	22	31	42%	56%	48%
Apache Struts	27	37	23	14	62%	85%	72%	32	12	20	38%	44%	41%
Apache Xerces	27	30	23	7	77%	85%	81%	74	23	51	31%	85%	46%
Overall	**651**	**782**	**552**	**230**	**71%**	**85%**	**77%**	**1,176**	**417**	**759**	**35%**	**64%**	**46%**

by both approaches m_i and m_j, and $correct_{m_i \setminus m_j}$ appraises the true smells detected by m_i only and missed by m_j. The latter metric provides an indication of how a code smell detection technique contributes to enriching the set of correct code smells identified by another approach. This information can be used to analyze the complementarity of structural and textual information when performing code smell detection.

9.3.2 Analysis of the Results

This section reports the results of our study, with the aim of addressing the research questions formulated in the previous section. To avoid redundancies, we report the results for all the three research questions together, discussing each smell

Table 9.3: Overlap between TACO and the structural techniques (ST). For Long Method and Blob the structural technique is DECOR, for Feature Envy it is JDeodorant, for Promiscuous Package it is the approach proposed in [196], for Misplaced Class the approach proposed in [76].

Code Smell	TACO∩ST		TACO\ST		ST\TACO	
	#	%	#	%	#	%
Long Method	364	60%	188	31%	53	9%
Feature Envy	101	46%	58	26%	62	28%
Blob	138	42%	138	42%	49	16%
Promiscuous Package	43	28%	78	51%	33	21%
Misplaced Class	12	21%	39	67%	8	12%

separately. Tables 9.2, 9.4, 9.5, 9.6, and 9.7 show the results achieved by TACO and by the structural approaches on the ten subject systems for *Long Method*, *Feature Envy*, *Blob*, *Promiscuous Package* and *Misplaced Class*, respectively. Specifically, we report (i) the number of actual components affected by a smell (column #*Actual Smells*), (ii) the number of components detected as smelly by the textual and structural approaches (column *Det*), (iii) the number of true and false positive instances identified by each approach (columns *TP* and *FP*), and (iv) accuracy metrics for each approach involved in a comparison, in terms of precision, recall, and f-measure. In addition, Table 9.3 reports values concerning overlap and differences between TACO and the structural techniques: column "TACO ∩ ST" reports the percentage of correct smell instances detected by both TACO and the alternative structural approach; column "TACO \ ST" reports the percentage of correct code smells correctly identified by TACO but missed by the structural technique; finally, column "ST \ TACO" reports the percentage of correct code smells identified by the structural technique but not by TACO. In the following, we discuss the results for each smell type.

Long Method Discussion

In the set of subject systems, we found 651 instances of this smell. The analysis of the results indicates that a textual approach more accurately detects instances of the Long Method smell. Specifically, the F-measure on the overall dataset of TACO is 77% (85% of recall and 71% of precision) against 46% (64% of recall and 35% of precision) achieved by the alternative approach. The low accuracy achieved by DECOR is due to the fact that the number of lines of code (LOC) of a method only tells a part of the whole story. Indeed, there are several cases in which a method is cohesive even though the large size of the method can indicate the presence of such a smell. The use of textual analysis is able to better discriminate whether a method implements more than one functionality. Of particular interest is the case of the `Eclipse Core` project: Here we found several methods having more than 100 LOC, implementing intrinsically complex operations, but not characterized by the presence of a Long Method smell. For example, the method `createSingleAssistNameReference` of the class `CompletionParser` needs to parse the actual content of the IDE workspace in order to automatically suggest to developers the methods she can use in her context. Although the method has 113 LOC, it can not be considered a *Long Method* smell, since it has a focused responsibility implemented across multiple lines. However, the DECOR rule detects this method as affected by the smell. Conversely, TACO correctly discards it from the candidate smell set. Moreover, our approach is able to identify different types of *Long Method* instances with respect to the ones a structural technique can identify. As an example, the method `findTypesAndPackages` of the class `CompletionEngine`, allows to discover the classes and the packages of a given project. Clearly, this method manages different tasks, even if its size is not large (*i.e.*, 58 lines of code). This means that the use of textual analysis is actually useful to avoid the identification of many false positive candidates, but also to detect instances of *Long Method* that the structural technique is not able to detect. This claim is supported by the results achieved when analyzing the overlap between TACO and DECOR (see Table 9.3). Indeed, we observed an overlap of 60%, *i.e.*,

60% of the actual *Long Method* instances are correctly detected by both TACO and DECOR, while it is interesting to note that TACO is able to correctly detect 31% of instances that DECOR is not able to detect. Finally, 9% of instances are correctly identified by DECOR and missed by TACO. An example of a *Long Method* instance correctly identified by DECOR and missed by TACO can be found in the class `RetrieveEngine` of the `Apache Ivy` project, where the method `retrieve` is characterized by 165 LOC. This method implements the basic operation of finding the settings of the machine Ivy is working on, however it also has an auxiliary function checking whether or not the settings are up to date. The textual approach fails in the detection of this smell because of the consistent vocabulary of the method. The achieved results highlight a tangible potential of combining structural and textual information for detecting this type of smell. We are planning to tackle this combination as part of our future work.

Feature Envy Discussion

For the *Feature Envy* smell, we found a total of 228 affected methods in our dataset. TACO has been able to identify 159 of them (recall of 70%), against the 163 detected by JDeodorant (recall of 71%). On the other hand, the precision obtained by TACO is higher than the one achieved by JDeodorant (69% against 61%). Overall, TACO's F-Measure is higher than JDeodorant (69% against 66%), and our approach outperforms the alternative one on 13 out of 20 systems (65% of the times). It is important to note that *JDeodorant* is a *refactoring* tool and, as such, it identifies *Feature Envy* smells with the purpose of suggesting opportunities of *Move Method refactoring*. Thus, the tool detects the smell only if the application of the refactoring is actually possible. To this aim, JDeodorant checks some preconditions to ensure that the program behavior does not change after the application of the refactoring [25]. For example, one of the preconditions considered is that *the envied class does not contain a method having the same signature as the moved method* [25]. In order to set a fair comparison with our approach, we filtered the *Feature Envy* instances found by our approach, using the same set of preconditions defined by JDeodorant [25]. During this step, we removed 1 correct instances and 3 false positives from the

Table 9.4: Feature Envy - TACO accuracy compared to JDeodorant.

	#Actual	TACO						DECOR					
Project	Smells	Det.	TP	FP	Prec.	Rec.	F-M	Det.	TP	FP	Prec.	Rec.	F-M
ArgoUML	5	4	3	1	75%	60%	67%	7	4	3	57%	80%	67%
Apache Ant	8	9	6	3	67%	75%	71%	13	2	11	15%	25%	19%
aTunes	8	10	6	4	60%	75%	67%	16	6	10	34%	75%	50%
Apache Cassandra	28	21	18	3	86%	64%	73%	28	28	0	100%	100%	100%
Eclipse Core	3	4	2	2	50%	67%	57%	0	0	0	0%	0%	0%
FreeMind	1	2	1	1	50%	100%	67%	5	0	5	0%	0%	0%
HSQLDB	14	14	9	5	64%	64%	64%	19	9	10	47%	64%	55%
Apache Hive	22	17	15	2	88%	68%	77%	19	17	2	89%	77%	83%
Apache Ivy	17	15	10	5	67%	59%	63%	13	10	3	77%	59%	67%
Apache Log4j	3	3	1	2	34%	34%	34%	9	2	7	22%	67%	34%
Apache Lucene	26	31	19	12	61%	73%	67%	45	23	22	51%	88%	65%
JEdit	10	8	6	2	75%	60%	67%	3	3	0	100%	30%	46%
JHotDraw	8	11	7	4	64%	88%	74%	9	6	3	67%	75%	71%
JVLT	1	2	1	1	50%	100%	67%	2	1	1	50%	100%	67%
Apache Karaf	14	18	13	5	72%	93%	81%	16	12	4	75%	86%	80%
Apache Nutch	11	9	6	3	67%	55%	60%	13	7	6	54%	64%	58%
Apache Pig	7	9	5	4	56%	71%	63%	7	4	3	57%	57%	57%
Apache Qpid	15	17	13	4	76%	87%	81%	15	11	4	73%	73%	73%
Apache Struts	19	17	12	5	71%	63%	67%	22	13	9	59%	68%	63%
Apache Xerces	8	9	6	3	67%	75%	71%	8	5	3	63%	63%	63%
Overall	**228**	**230**	**159**	**71**	**69%**	**70%**	**69%**	**269**	**163**	**106**	**61%**	**71%**	**66%**

initial set. Once the filtering has been applied, TACO's precision increases to 70%, while its recall decreases to 69%. Moreover, it is interesting to note that the two approaches are highly complementary, as reported in Table 9.3. In fact, 46% of the correct smell instances have been detected by both approaches, while our textual technique identifies 26% of instances missed by JDeodorant. On the other side, the structural tool is able to capture 28% of correct *Feature Envy* instances missed by TACO. An example of an instance correctly captured by TACO is represented by the method `readSchema` of the class `IndexSchema` of `Apache Lucene`. Here the method, implementing the functionality able to read the schema of a database, has a concept more related to the class `ZkIndexSchemaReader` with respect to the class it is actually in. On the other hand, JDeodorant is the only technique able to correctly identify the smell affecting the method `isRebuildRequired` of the

Table 9.5: Blob - TACO accuracy compared to DECOR.

	#Actual	TACO						DECOR					
Project	Smells	Det.	TP	FP	Prec.	Rec.	F-M	Det.	TP	FP	Prec.	Rec.	F-M
ArgoUML	30	42	28	14	67%	93%	77%	23	15	8	65%	50%	57%
Apache Ant	31	37	25	12	68%	81%	74%	21	17	4	81%	55%	65%
aTunes	9	11	7	4	64%	78%	70%	4	3	1	75%	34%	46%
Apache Cassandra	22	26	20	6	77%	91%	83%	5	4	1	80%	18%	30%
Eclipse Core	43	56	35	21	63%	81%	71%	64	31	33	48%	72%	52%
FreeMind	11	12	9	3	75%	82%	78%	7	7	0	100%	64%	78%
HSQLDB	23	24	18	6	75%	78%	77%	32	18	14	57%	78%	66%
Apache Hive	27	24	17	7	71%	63%	67%	18	13	5	72%	48%	58%
Apache Ivy	10	10	8	2	80%	80%	80%	3	3	0	100%	30%	46%
Apache Log4j	5	6	4	2	67%	80%	73%	3	1	2	34%	20%	25%
Apache Lucene	27	30	22	8	74%	81%	77%	27	18	9	67%	67%	67%
JEdit	15	13	12	1	92%	80%	86%	13	11	2	85%	74%	79%
JHotDraw	13	14	11	3	79%	85%	81%	11	8	3	73%	62%	67%
JVLT	3	2	1	1	50%	34%	40%	1	1	0	100%	34%	50%
Apache Karaf	5	4	3	1	75%	60%	67%	4	3	1	75%	60%	67%
Apache Nutch	2	2	1	1	50%	50%	50%	2	1	1	50%	50%	50%
Apache Pig	7	6	4	2	67%	57%	62%	12	5	7	42%	71%	53%
Apache Qpid	29	34	27	7	79%	93%	86%	15	12	3	80%	41%	55%
Apache Struts	13	17	11	6	65%	85%	73%	6	4	2	67%	31%	42%
Apache Xerces	16	20	14	6	70%	88%	78%	22	13	9	59%	81%	68%
Overall	**338**	**388**	**276**	**112**	**71%**	**82%**	**76%**	**292**	**187**	**105**	**64%**	**55%**	**59%**

class `WebsphereDeploymentTool`, present in `Apache Ant` project. In this case, TACO is not able to identify the smell since it is characterized by a high number of dependencies with the envied class, even if the textual content of the method is more related to the actual class.

Blob Discussion

As for the detection of *Blobs*, TACO is able to achieve, overall, a precision of 71% and a recall of 82% (F-measure=76%), while DECOR is able to achieve a precision of 64% and a recall of 55% (F-measure=59%). Specifically, on average, TACO is 16% more accurate in detecting this type of smell. The only exception is represented by the `jVLT` project. In this case, DECOR is able to identify one out of three

Blob instances present in the system without any false positive (precision=100%, recall=34%), while TACO outputs two candidates, of which one is a false positive (precision=50%, recall=34%). In particular, TACO fails in the suggestion of the class `Utils` of the package `net.sourceforge.jvlt.utils`. Even if the methods of this class are not cohesive, since the class implements several utility methods used by other classes, it can not be considered a Blob since it does not centralize the behavior of a portion of the system. On the contrary, an example of *Blob* correctly detected by TACO can be found in the class `AudioFile` of the `aTunes` project. This class has the goal to map an entity, but actually it implements several methods for the management of such entities and also for the management of users' playlists. DECOR can not detect this smell since the class does not seem to be a *controller* class[6]. Looking at the complementarity in Table 9.3, we observed that the textual approach is able to detect a consistent number of correct instances missed by DECOR. Indeed, TACO is able to capture 42% of classes affected by *Blob* missed by DECOR, while DECOR detects 16% of instances missed by TACO. Noticeably, 42% of correct instances are identified by both the approaches. This result highlights how the use of textual analysis can be particularly suitable for detecting the *Blob* code smell.

Promiscuous Package Discussion

Over the set composed of 163 *Promiscuous Package* instances, TACO achieves 67% of precision and 74% of recall (F-measure=71%), outperforming on 19 systems out of 20 the alternative structural-based technique. This result clearly indicates how the use of textual information is beneficial in order to identify packages composed of classes implementing different responsibilities. The only exception regards the `aTunes` project, in which the two techniques obtain the same accuracy (F-Measure=40%). Specifically, in this system there are 3 promiscuous packages, and the two approaches are able to correctly detect only one instance each. TACO detects as promiscuous the `atunes.kernel.actions` package, that is charac-

[6]DECOR identify a controller class if its name contains a suffix in a set {*Process, Control, Command, Manage, Drive, System*}

Table 9.6: Promiscuous Package - TACO accuracy compared to the approach proposed in [196].

	#Actual	TACO						DECOR					
Project	Smells	Det.	TP	FP	Prec.	Rec.	F-M	Det.	TP	FP	Prec.	Rec.	F-M
ArgoUML	13	18	11	7	61	85	71	13	8	5	62	62	62
Apache Ant	10	9	8	1	89	80	84	6	4	2	67	40	50
aTunes	3	2	1	1	50	33	40	2	1	1	50	33	40
Apache Cassandra	7	8	5	3	63	71	67	5	3	2	60	43	50
Eclipse Core	9	10	7	3	70	78	74	6	3	3	50	33	40
Freemind	6	7	4	3	57	67	62	3	2	1	67	33	44
HSQLDB	7	7	5	2	71	71	71	5	3	2	60	43	50
Apache Hive	7	6	5	1	83	71	77	7	4	3	57	57	57
Apache Ivy	4	2	2	0	100	50	67	2	1	1	50	25	33
Apache Log4j	5	3	3	0	100	60	75	2	2	0	100	40	57
Apache Lucene	26	34	24	10	71	92	80	27	19	8	70	73	72
JEdit	7	8	5	3	63	71	67	5	2	3	40	29	33
JHotdraw	5	6	3	3	50	60	55	3	1	2	33	20	25
JVLT	3	4	1	3	25	33	29	2	0	2	00	00	0
Apache Karaf	5	5	3	2	60	60	60	4	2	2	50	40	44
Apache Nutch	4	5	2	3	40	50	44	4	1	3	25	25	25
Apache Pig	10	9	8	1	89	80	84	6	5	1	83	50	63
Apache Qpid	15	17	11	6	65	73	69	12	7	5	58	47	52
Apache Struts	13	16	11	5	69	85	76	11	7	4	64	54	58
Apache Xerces	4	4	2	2	50	50	50	3	1	2	33	25	29
Overall	**163**	**180**	**121**	**59**	**67%**	**74%**	**71%**	**128**	**76**	**52**	**59%**	**47%**	**52%**

terized by 133 classes implementing actions related to the management of (i) the buttons present in the UI, (ii) the options for saving audio files in local, and (iii) the synchronization of the playlists. In this case, the structural technique can not detect the instance because of the dependencies among the classes, due to the fact that all the classes inherit the `AbstractAction` class and belong to the menu visible by the end user.

On the other hand, TACO fails in the detection of the `atunes.gui.views` package, that is composed of 26 classes related to different aspects of the management of the dialogs of the application. The structural technique correctly detects the smell since it is able to cluster the classes into sub-packages, while TACO can

not detect it because of the consistent vocabulary used in the classes. Looking at Table 9.3, we can see that the textual technique captures the most part of the instances missed by the alternative approach (*i.e.*, 51%), while the structural technique detect 21% of instances missed by TACO. Finally, it is interesting to note how only the 28% of instances are detected by both the techniques.

Table 9.7: Misplaced Class - TACO accuracy compared to the approach proposed by Atkinson and King [76].

	#Actual	TACO						DECOR					
Project	Smells	Det.	TP	FP	Prec.	Rec.	F-M	Det.	TP	FP	Prec.	Rec.	F-M
ArgoUML	-	-	-	-	-	-	-		-	-	-	-	-
Apache Ant	4	4	3	1	75	75	75	4	2	2	50	50	50
aTunes	-	-	-	-	-	-	-		-	-	-	-	-
Apache Cassandra	-	-	-	-	-	-	-		-	-	-	-	-
Eclipse Core	11	11	8	3	73	73	73	9	7	2	78	64	70
Freemind	-	-	-	-	-	-	-		-	-	-	-	-
HSQLDB	1	2	0	2	0	0	0	3	1	2	33	10	50
Apache Hive	2	4	2	2	50	10	67	5	1	4	20	50	29
Apache Ivy	2	4	2	2	50	10	67	6	1	5	17	50	25
Apache Log4j	-	-	-	-	-	-	-		-	-	-	-	-
Apache Lucene	11	11	9	2	82	82	82	7	5	2	71	45	56
JEdit	-	-	-	-	-	-	-		-	-	-	-	-
JHotdraw	4	3	2	1	67	50	57	3	2	1	67	50	57
JVLT	-	-	-	-	-	-	-		-	-	-	-	-
Apache Karaf	1	4	1	3	25	10	40	3	0	3	0	0	0
Apache Nutch	-	-	-	-	-	-	-		-	-	-	-	-
Apache Pig	21	21	19	2	90	90	90	4	3	1	75	14	24
Apache Qpid	2	3	2	1	67	10	80	2	0	2	0	0	0
Apache Struts	-	-	-	-	-	-	-		-	-	-	-	-
Apache Xerces	4	4	3	1	75	75	75	3	2	1	67	50	57
Overall	**63**	**71**	**51**	**20**	**72%**	**81%**	**76%**	**49**	**24**	**25**	**49%**	**38%**	**43%**

Misplaced Class Discussion

We found 63 instances of *Misplaced Class* over 11 of the 20 systems analyzed. Overall, the textual technique reaches 72% of precision and 81% of recall (F-measure=76%),

while the alternative approach has a precision of 49%, with a recall of 38% (F-measure=43%). This result clearly shows the usefulness of textual analysis for detecting classes not properly located. An example smell detected by TACO can be found in the `Apache Lucene` project, where the class `InstantiatedIndex` of the package `lucene.store` has different dependencies with the current package, but has topics more related to the package `lucene.index`. In contrast, the approach proposed in [76] is the only one able to detect, in the `Apache Pig` project, the `PhyPlanSetter` class of the package `mapReduceLayer` as misplaced. Here, the class has a vocabulary more similar to the package where it is actually in, but it should clearly be placed in the `physicalLayer` package. When considering the overlap metrics (Table 9.3), we confirm the actual superiority of TACO with respect the structural technique. Indeed, we found that 67% of correctly detected instances are only found by TACO, 21% of instances are detected by both approaches, while a smaller percentage (*i.e.,* 12%) of smells are only identified by the alternative structural approach.

9.4 Threats to Validity

This section describes the threats that can affect the validity of our empirical study.

Construct Validity. Threats to construct validity are mainly due to the definition of the *oracle* for the studied software projects. In the context of our work, we rely on the oracles publicly available in [60]. However, we cannot exclude that the oracle we used misses some smells, or else identified some false positives. Another threat is the use of comments during the detection process, since not all the systems have them. To deal with this, we re-run the case study by just considering identifiers. Results are in line with the ones obtained in Section 9.3.2. The interested reader can find detailed results in our online appendix [205]. Finally, another threat is related to our re-implementation of both *SEGMENT* [203] and *DECOR* [20], which was needed because of lack of tools. However, our re-implementations use the exact algorithms defined by Wang *et al.* [203] and by Moha *et al.* [20], and have

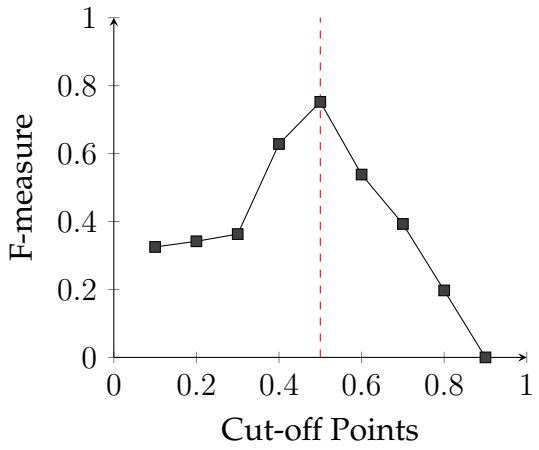


Figure 9.2: F-measure achieved with different cut-off points. Dashed red line corresponds to our cut-off point.

already been used in earlier work [52, 47, 206].

Internal Validity. An important factor that might affect the results achieved when evaluating the accuracy of TACO is represented by the cut-off point we used to detect code smells. To have higher reliability of our choice, we also investigated the effects of different cut-off points on the performance of TACO when detecting smells for all the systems considered in our study. For example, Figure 9.2 plots the F-measure scores achieved by TACO when using different cut-off points when detecting *Long Method* on the `aTunes` project. We can notice that the best F-measure is achieved when using as cut-off point the median of the probability distribution, *i.e.*, the dashed red line in Figure 9.2. Similar results are also obtained for the other projects and smell types, as reported in our online appendix [205].

Another threat to internal validity is represented by the settings used for the IR process. During the pre-processing, we filtered the textual corpus by using well known standard procedures: stop word list, stemmer and the *tf-idf* weighting schema, and identifiers splitting [55]. For LSI, we choose the number of concepts (k) using the heuristics proposed by Kuhn *et al.* [200].

External Validity. We demonstrated the feasibility of our approach focusing our attention on smells of different nature and of different levels of granularity. How-

ever, there might be other smells that can be potentially detected using TACO and not considered in this chapter [8, 61]. Such an investigation is part of our future agenda. Another threat can be related to the number of subject systems used in our empirical evaluation. To show the generalizability of our results, we conducted an empirical study involving 20 Java open source systems having different size and different domains. It could be worthwhile to replicate the evaluation on other projects written in different programming languages.

9.5 Conclusion

In this chapter, we devised TACO (**T**extual **A**nalysis for **C**ode smell detecti**O**n), an approach purely based on textual analysis able to detect instances of five different code smell types having different nature and different granularity, *i.e., Long Method*, *Feature Envy*, *Blob*, *Promiscuous Package*, and *Misplaced Class*. It does so by analyzing the properly decomposed textual blocks composing a code component, in order to apply IR methods and measuring the probability that a component is affected by a given smell.

To evaluate the accuracy of the proposed detector, we ran TACO on 20 open source software systems in order to (i) evaluate its accuracy in detecting code smells, and (ii) compare its performances with state-of-the-art structural-based approaches. The results of the study demonstrated that TACO exhibits a precision ranging between 67% and 72%, and a recall that ranges between 70% and 85%, often outperforming alternative structural-based detectors. Moreover, we found some complementarity between textual and structural information, suggesting that their combination could be beneficial to obtain better detection accuracy.

For the aforementioned reason, our future research agenda includes the definition of a combined technique, as well as the evaluation of the usefulness of the proposed textual approach in the detection of other code smell types. Also, we plan to define a smell prioritization approach using the smelliness probability computed by TACO. Finally, we will further investigate the characteristics of

textual smells in order to compare their impact on change- and fault-proneness with the results achieved in previous studies considering smells detected using structural techniques.

Chapter 10

The Scent of a Smell: an Extensive Comparison between Textual and Structural Smells

10.1 Introduction

Technical debt is a metaphor introduced by Cunningham in 1993 to indicate *"not quite right code which we postpone making it right"* [4]. The metaphor tries to explain the compromise between delivering the most appropriate but still immature product, in the shortest time possible [4, 130, 131, 132, 133]. *Bad code smells* ("code smells" or "smells"), *i.e.,* symptoms of poor design and implementation choices applied by programmers during the development of a software project [8], represent an important factor contributing to technical debt [131].

The research community spent a lot of effort studying the extent to which code smells tend to remain in a software project for long periods of time [12, 13, 14, 15], as well as their negative impact on non-functional properties of source code, such as program comprehension [18], change- and defect-proneness [16, 17], and, more generally, maintainability [9, 117, 118]. As a consequence, several tools and techniques have been proposed to help developers in detecting code smells and to suggest refactoring opportunities [20, 21, 25, 27, 30, 52]. So far, almost all detectors

try to capture code smell instances using structural properties of source code as the main source of information.

Nevertheless, recent studies have indicated that code smells detected by existing tools are generally ignored (and thus not refactored) by the developers [14, 40, 49]. A possible reason is that developers do not perceive the smells identified by the tool as actual design problems or, if they do, they are not able to practically work on such code smells. In other words, there is misalignment between what is considered smelly by the tool and what is actually refactorable by developers. In this chapter, we conjecture that *rather than structural properties, developers consider textual information more close to the way they reason and act on code smell instances, and find them more useful during refactoring*.

To this aim, we conducted two different studies aimed at investigating how developers **act on** instances of the same type of smell but detected using structural tools versus an approach based purely on textual information, named TACO (see Chapter 9). First, we performed a software repository mining study considering 301 releases and 183,514 commits from 20 open source projects in order (i) to verify whether textually and structurally detected code smells are treated and refactored differently, and (ii) to analyze their likelihood of being resolved with regards to different types of code changes, *e.g.*, refactoring operations. Since our quantitative study cannot explain relation and causation between types of smell and maintenance activities, we perform a qualitative study with 10 industrial developers and 5 software quality experts in order to understand (i) how code smells identified using different sources of information are perceived, and (ii) whether textually or structurally detected smells are easier to refactor. In both the two studies, we focused on five code smell types, *i.e., Blob*, *Feature Envy*, and *Long Method*, *Misplaced Class*, and *Promiscuous Package*.

The results of our study indicate that textually detected code smells are perceived as harmful as the structural ones, even though they do not exceed any typical software metrics' value (*e.g.,* lines of code in a method). Moreover, source code with inconsistent vocabulary is easier to comprehend and, therefore, easier to maintain and refactor. We also observed that developers are more able to

Table 10.1: The Code Smells considered in our Study

Name	Description
Blob	A large class implementing different responsibilities.
Feature Envy	A method is more interested in a class other than the one it actually is in.
Long Method	A method that implements more than one function.
Misplaced Class	A class that should be placed in another package.
Promiscuous Package	A package composed of unrelated classes.

recognize design flaws affecting textual smells. As a consequence, developers' activities tend to decrease the intensity of textual code smells, positively impacting their likelihood of being resolved. Vice versa, structural smells typically increase in intensity over time, indicating that maintenance operations are not aimed at removing them or do not limit them.

10.2 Textual and Structural Code Smell Detection

Starting from the definition of design defects proposed in [8, 61, 62, 63], researchers have devised tools and techniques to detect code smells in software systems. Most of them are based on the analysis of the properties extractable from the source code (*e.g.*, method calls) by means of a combination of structural metrics [20, 21, 22, 23, 24, 26, 65, 72, 103], while in recent years the use of alternative sources of information (*i.e.*, historical and textual analysis) have been explored [52, 53], together with methodologies based on machine learning [36, 37] and search-based algorithms [32, 33, 34, 35].

Besides code smell detectors, even refactoring techniques may be adopted to identify design flaws in the source code [25, 27, 29, 31, 73, 66]. Rather than identify code smells directly, such approaches recommend refactoring operations to apply in order to remove a design flaw. Also in this case, the primary source of information exploited is the structural one [25, 73, 66], while few works have explored a combination of structural and conceptual analysis [27, 31, 29].

Both code smell detectors and refactoring techniques defined so far mostly fo-

cused their attention on the design flaws object of this study, *i.e.*, *Blob*, *Long Method*, *Feature Envy*, *Misplaced Class*, and *Promiscuous Package*. Table 10.1 briefly describes each of them. The interest in these smells is dictated by the fact that they have been widely recognized as important threats to the maintainability of a software system [9, 10, 16, 17, 18, 19], but also because they are considered by the developers as harmful [50]. In our study, we consider a smell *textual* when it is detected using a textual-based detection technique, *i.e.*, it is characterized by high textual scattering among the elements it contains (*e.g.*, textual content of methods or statements). On the other hand, a smell is *structural* if it is detected by a detector purely based on the analysis of structural properties of source code (*e.g.*, number of attributes, size or number of dependencies with other classes). The following subsections describe the detection rules applied in the context of our empirical study.

10.3 Textual Smells Detection

As for the detection of *textual* code smells, we use **TACO** (**T**extual **A**nalysis for **C**ode smell de**T**ection) [53], an approach able to identify code smells using a three-step process, *i.e.*, (i) textual content extraction, (ii) application of IR normalization process, and (iii) application of specific heuristics in order to detect code smells related to promiscuous responsibilities (*e.g.*, Blob). In the first step TACO extracts all textual elements needed for the textual analysis process of a software project, i.e., source code identifiers and comments. Then, the approach applies a standard IR normalization process [55] aimed at (i) separating composed identifiers, (ii) reducing to lower case letters the extracted words, (iii) removing special characters, programming keywords and common English stop words, and (iv) stemming words to their original roots via Porter's stemmer [197]. Thus, the code smell detection process relies on Latent Semantic Indexing (LSI) [204], an extension of the Vector Space Model (VSM) [55], that models code components as vectors of terms occurring in a given software system. LSI uses Singular Value Decomposition (SVD) [199] to cluster code components according to the relationships among words and among code components (co-occurrences). The original vectors (code com-

ponents) are then projected into a reduced *k* space of concepts to limit the effect of textual noise. To this aim, TACO uses the well-known heuristic proposed by Kuhn *et al.* [200], i.e., $k = (m \times n)^{0.2}$ where m denotes the vocabulary size and n denotes the number of documents (code components). Finally, code smells are detected by measuring the lack of textual similarity among their constituent code components (*e.g.,* vectors) using the cosine distance. For example, a *Blob* is detected (i) by computing the average similarity among the methods of the class, which correspond to the textual cohesion of a class defined by Marcus and Poshyvanyk [45]; and (ii) by reversing the textual cohesion in order to obtaine a probability P_B that a class is affected by the *Blob* code smell.

Using the same steps, TACO is able to detect *Long Method* instances. Indeed, it firstly extract the code blocks composing a method by using the approach by Wang *et al.* [203], and then TACO computes the probability a method is smelly considering the lack of cohesion among the code blocks composing it. Instances of *Promiscuous Package* are instead detected by exploiting the lack of cohesion among the classes composing a package.

As for the *Feature Envy* code smell, for each method M belonging to the class C_O, the approach retrieves the more similar class ($C_{closest}$), and then the probability that the method is smelly is given by the difference between the textual similarities of M and the two classes $C_{closest}$ and C_O. Similarly, TACO identifies *Misplaced Class* instances by retrieving the package $P_{closest}$ (*i.e.*, the more similar package) for a class C contained in the package P_O, and then computing the probability that this class is misplaced by measuring the difference between the textual similarities of C and the two packages $P_{closest}$ and P_O.

For all types of code smells, TACO outputs a value ranging in $[0; 1]$, which indicates the probability of a code component to be affected by a specific code smell.

10.4 Structural Smells Detection

Regarding the detection of structural smells, we rely on *DECOR*, the structural approach proposed by Moha *et al.* [20] for detecting *Blob* and *Long Method* smells. In particular, this approach uses a set of rules, called rule cards[1], describing the characteristics a code component should have in order to be classified as smelly. In practice, rules are set of conditions based on code metrics (*e.g.,* line of codes) with respect to fixed thresholds. For example, a Blob is detected when a class has an LCOM5 (Lack of Cohesion Of Methods) [64] higher than 20, a number of methods and attributes higher than 20, and it has a *one-to-many* association with data classes. As for *Long Method*, DECOR classifies a method as smelly if it has more than 100 lines of code. For detecting *Feature Envy* instances, we used *JDeodorant* [25]. Given a method m, *JDeodorant* forms a set of candidate target classes where m could be moved to, based on its structural relationships. Then, the candidate target classes are sorted in descending order according to the number of dependencies m has to each of them. If the movement of m in the first ranked class satisfies a set of behavior preserving preconditions (e.g., the target class does not contain a method with the same signature as m) then m is marked as a *Feature Envy* [25].

For the *Misplaced Class* detection, we re-implemented the approach proposed by Atkinson and King [76], which traverse the abstract syntax tree of a class C in order to determine, for each feature, the set T of classes referencing them. Then, the classes in T are sorted based on their belonging package in order to extract the number of dependencies each package $P \in T$ has with the class C. If C has more dependencies with a different package with respect to the one it is actually in, an instance of *Misplaced Class* is detected. Our re-implementation relies on the publicly available Java Development Tools APIs[2].

Finally, to detect *Promiscuous Package* instances, we re-implemented the approach by Girvan *et al.* [196]. It is based on a clustering algorithm that groups together classes of a package based on the dependencies among them. In the re-

[1] `http://www.ptidej.net/research/designsmells/`

[2] http://www.eclipse.org/jdt/

implementation, we exploited the X-Means algorithm [207], an extension of the traditional K-Means [208] where the parameter X (*i.e.*, the number of clusters the algorithm must form) is automatically configured using a heuristic based on the Bayesian Information Criterion [207]. If the algorithm finds more than one cluster, it means that the classes contained in the package under analysis contain unrelated responsibilities and, therefore, an instance of the *Promiscuous Package* smell is detected.

10.5 Study I: The Evolution of Textual and Structural Code Smells

Table 10.2: Characteristics of the Software Projects in Our Dataset

System	#Releases	#Commits	Classes	Methods	KLOCs
ArgoUML	16	19,961	777-1,415	6,618-10,450	147-249
Apache Ant	22	13,054	83-813	769-8,540	20-204
aTunes	31	6,276	141-655	1,175-5,109	20-106
Apache Cassandra	13	20,026	305-586	1,857-5,730	70-111
Eclipse Core	29	21,874	744-1,181	9,006-18,234	167-441
FreeMind	16	722	25-509	341-4,499	4-103
HSQLDB	17	5,545	54-444	876-8,808	26-260
Apache Hive	8	8,106	407-1,115	3,725-9,572	64-204
Apache Ivy	11	601	278-349	2,816-3,775	43-58
Apache Log4j	30	2,644	309-349	188-3,775	58-59
Apache Lucene	6	24,387	1,762-2,246	13,487-17,021	333-466
JEdit	29	24,340	228-520	1,073-5,411	39-166
JHotDraw	16	1,121	159-679	1,473-6,687	18-135
JVLT	15	623	164-221	1,358-1,714	18-29
Apache Karaf	5	5,384	247-470	1,371-2,678	30-56
Apache Nutch	7	2,126	183-259	1,131-1,937	33-51
Apache Pig	8	2,857	258-922	1,755-7,619	34-184
Apache Qpid	5	14,099	966-922	9,048-9,777	89-193
Apache Struts	7	4,297	619-1,002	4,059-7,506	69-152
Apache Xerces	16	5,471	162-736	1,790-7,342	62-201
Overall	**301**	**183,514**	**25-2,246**	**188-17,021**	**4-466**

In this study we empirically investigate how developers deal with textually and structurally detected code smells by applying software repository mining techniques.

10.5.1 Empirical Study Definition and Design

The *goal* of the empirical study is to evaluate the impact of different sources of information on developers' notion of code smells. Our *conjecture* is that code smells characterized by an inconsistent vocabulary are easier to perceive and/or easier to remove for developers when compared to code smells characterized by structural design flaws, such as high number of dependencies or large size, since conceptual aspects of source code can provide direct insight that a developer can use to understand and work on code components affected by design problems. The *context* of the study consists of the five code smells presented in the previous Section. Moreover, we conduct our analyses on twenty open source software projects. Table 10.2 reports the characteristics of the analyzed systems, namely the number of public releases, and their size in terms of number of commits, classes, methods, and KLOC. Among the analyzed projects, we have twelve projects belonging to the Apache ecosystem[3], and eight open source projects from elsewhere. Note that our choice of the subject systems is not random, but instigated by our aim to analyze projects belonging to different ecosystems, having different size and scope.

Our investigation aims at answering the following research questions:

- **RQ1:** *Are textually or structurally detected code smells more likely to be resolved?*
- **RQ2:** *Do structural or textual smells evolve differently with regard to different types of changes (Bug fixing, Enhancement, New feature, Refactoring)?*

To answer **RQ1**, we first manually detect the releases (both major and minor ones) of the software projects in our dataset, identifying 301 of them spread across the 20 subject systems. Then, our CHANGEHISTORYMINER tool analyzes each release R of a software project p_i to detect code components (*i.e.*, methods or classes) affected by one of the considered code smells. Specifically, our tool mines each release of p_i, checks out the corresponding version of the source code, and runs the textual and structural detection rules presented in Section 10.2.

[3] `http://www.apache.org/` verified October 2016

To monitor the evolution of code smells, a simple truth value representing the presence or absence of a design flaw is not enough because we might not evaluate how the severity of structurally and textually detected code smells varies (decreases/increases) over the releases of the projects in our dataset. Hence, once a code smell is detected we monitor its evolution in terms of *intensity*, *i.e.,* in terms of variation of the degree of severity of a code smell.

Computing the *intensity* is easy for TACO, since it outputs a value $\in [0; 1]$ indicating the probability that a code component is affected by a code smell. Instead, for structural smells we need to modify the detection approach. In particular, DECOR classifies a code component as smelly if and only if a set of conditions (rules) are satisfied, where each condition has the form $\texttt{if metric}_i \geq \texttt{threshold}_i$. Therefore, the higher the distance between the actual code metric ($\texttt{metric}_i$) and the fixed threshold value ($\texttt{threshold}_i$), the higher the *intensity* of the flaw. Thus, if a class is detected as *Blob* by DECOR, we measure its intensity as follows: (i) we computed the differences between its actual values of software metrics (*e.g.,* LCOM5, number of methods, etc.) with respect to the corresponding thresholds reported in the rule card [20]; (ii) we normalized the obtained difference scores in $[0; 1]$ (iii) we measure the final intensity as the mean of those normalized scores.

As *Long Method*s are detected by only looking at the LOC (lines of code), the intensity is measured as the normalized difference between the LOC in a method and its threshold in the rule card, which is 100.

JDeodorant marks a method m as *Feature Envy* if and only if it has more structural dependencies with another class C^* with respect to the number of dependencies m has with the original class C (and if all preconditions are preserved). Therefore, the *intensity* is given by the normalized difference of the number of dependencies with C^* (new class) and the number of dependencies with C (original class).

The same strategy can be applied to measure the *intensity* of *Misplaced Class* instances. Indeed, as the technique by Atkinson and King [76] identifies this smell by looking at the difference between the dependencies a class C has toward a

Table 10.3: Tags assigned to commits involving smells.

Tag	Description
Bug fixing	The commit aimed at fixing a bug
Enhancement	The commit aimed at implementing an enhancement in the system
New feature	The commit aimed at implementing a new feature in the system
Refactoring	The commit aimed at performing refactoring operations

package P^* and the dependencies C has with the original package P, the *intensity* is given by the normalized difference between them.

Finally, we measure the *intensity* of *Promiscuous Package* smell by applying a *min-max* normalization on the number of clusters of classes found by the approach for a package P. In this way, the higher the number of clusters detected the higher the proneness of the package to be promiscuous.

From this monitoring, we obtained two distributions for each type of code smells: one distribution related to the variation of intensity for textual smells (Δ_{text}) over the different releases and a second one regarding the variation of intensity for structural smells (Δ_{struct}) over the same releases. Negative values for Δ_{text} (or Δ_{struct}) indicate that the intensity of textual (or structural) smells decreases over time, while positive values indicate that the intensity increases over time. In order to verify whether the differences (if any) between Δ_{text} and Δ_{struct} are statistically significant, we used the non-parametric Wilcoxon Rank Sum test [142] with ρ-value = 0.05. We also estimated the magnitude of the observed differences using Cliff's Delta (or d), a non-parametric effect size measure [143] for ordinal data. We follow the guidelines in [143] to interpret the effect size values: small for $d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d <$ 0.474 and large for $d \geq 0.474$.

As for **RQ2**, we are interested in understanding whether particular types of changes made by developers have a higher impact on the increase/decrease of the intensity of code smells. To this aim, we conducted a *fine-grained* analysis, investigating all the commits available in the repositories of the involved projects

(overall, we mined 183,514 commits) in order to understand *what type of action the developer was doing when modifying smelly classes*. Given a repository r_i, our mining tool `ChangeHistoryMiner` mines the entire change history of r_i, and for each commit involving a code smell runs the mixed technique proposed by Tufano *et al.* [47] in order to detect the types of changes shown in Table 10.3, *i.e., Bug Fixing*, *Enhancement*, *New Feature*, and *Refactoring*. To this aim, we download the issues for all 20 software projects from their BUGZILLA or JIRA issue trackers. Then, we check if a commit involving a textual or structural code smell is actually related to any collected issues. To link issues to commits, the approach by Tufano *et al.* complements two distinct approaches: the first one is based on regular expressions [144], which match the issue ID in the commit note, while the second one is *ReLink*, the approach proposed by Wu *et al.* [145], which considers several constraints, *i.e.,* (i) the existence of a match between the committer and the contributor who created the issue in the issue tracking system, (ii) the time interval between the commit and the last comment posted by the same contributor in the issue tracker is less than seven days, and (iii) the cosine similarity between the commit note and the last comment referred above, computed using the Vector Space Model (VSM) [55], is greater than 0.7. When it was possible to find a link between a commit and an issue, and the issue has a type included in the catalogue of tags shown in Table 10.3, then the commit is automatically classified. In the other cases, we assign the tags using a semi-automatic process. Specifically, we use a keyword-based approach for detecting a commit's goal similar to the one presented by Fischer *et al.* [144], and then we manually validate the tags assigned by analyzing (i) the commit message and (ii) the unix diff between the commit under analysis and its predecessor. Overall, we tagged 27,769 commits modifying instances of textually and structurally detected code smells. For 18,276 of them, we found the tag automatically, while the remaining 9,493 were manually assigned[4]. Once we obtained the tagged commits, we investigated how the different types of code changes (independent variables) impact the variation of intensity of textual and structural code smell (dependent variable). In particular, for each object project and for each kind of code

[4]The thesis' author was responsible of this task.

smell we applied logistic regression models [162] using the following equation:

$$\pi(\text{BF}, \text{E}, \text{NF}, \text{R}) = \frac{e^{C_0 + C_1 \cdot \text{BF} + C_2 \cdot \text{E} + C_3 \cdot \text{NF} + C_4 \cdot \text{R}}}{1 + e^{C_0 + C_1 \cdot \text{BF} + C_2 \cdot \text{E} + C_3 \cdot \text{NF} + C_4 \cdot \text{R}}} \quad (10.1)$$

where the independent variables are the number of *Bug Fixing* (BF), *Enhancement* (E), *New Feature* (NF) and *Refactoring* (R) operations applied by developers during the time period between two subsequent releases; the (dichotomous) dependent variable is whether the intensity increases/decreases between two subsequent versions; and C_i are the coefficients of the logistic regression model. Then, for each model we analyze (i) whether each independent variable is significantly correlated with the dependent variable as estimated by the Spearman rank correlation coefficient (we consider a significance level of $\alpha = 5\%$), and (ii) we quantify such a correlation using the Odds Ratio (OR) [185] which, for a logistic regression model, is given by e^{C_i}. Odd ratios indicate the increase in likelihood of a code smell intensity increase/decrease as a consequence of a one-unit increase of the independent variable, e.g., number of bug fixing (BF). For example, if we found that *Refactoring* has an OR of 1.10 with textual Blobs, this means that each one-unit increase of the *Refactoring* made on a textual Blob mirrors a 10% higher chance for the Blob of being involved in a decrease of its intensity.

Overall, the data extraction to answer **RQ1** and **RQ2** took five weeks on 4 Linux machines having dual-core 3.4 GHz CPU (2 cores) and 4 Gb of RAM.

10.5.2 Analysis of the Results

Table 10.4 reports the mean and the standard deviation scores of the variation of intensity for textual (Δ_{text}) and structural (Δ_{struct}) code smells, collected for *Blob*, *Feature Envy*, *Long Method*, *Misplaced Class*, and *Promiscuous Package* instances. The results clearly indicate that textual smells are treated differently than structural ones: in most cases the intensity of textual smells tends to decrease over time, *i.e.*, the Δ_{text} values are negative; vice versa, the intensity of structural smells tends to increase over time, as indicated by the positive Δ_{struct} scores. For example, blobs in `JVLT` detected by structural tools have an average Δ_{struct}=0.86, *i.e.*, their structural

Table 10.4: Mean and Standard Deviations of Δ_{text} and Δ_{struct} of our dataset. Decreasing variations are reported in **bold** face. TS = Textual Smells; SS = Structural Smells.

Project	Blob				Feature Envy				Long Method				Misplaced Class				Promiscuous Package			
	TS		SS		TS		SS		TS		SS		TS		SS		TS		SS	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
ArgoUML	**-0.08**	±0.23	0.13	±0.33	**-0.03**	±0.20	0.10	±0.26	**-0.03**	±0.17	0.11	±0.22	-	-	-	-	**-0.12**	±0.25	0.38	±0.24
Apache Ant	**-0.11**	±0.27	0.18	±0.47	**-0.05**	±0.22	0.11	±0.43	**-0.04**	±0.20	0.09	±0.38	**-0.05**	±0.12	0.04	±0.39	**-0.07**	±0.13	0.37	±0.24
aTunes	**-0.08**	±0.24	0.11	±0.25	**-0.01**	±0.27	0.10	±0.41	**-0.06**	±0.28	0.08	±0.42	-	-	-	-	**-0.01**	±0.11	0.49	±0.33
Apache Cassandra	**-0.08**	±0.25	0.18	±0.35	**-0.06**	±0.26	0.21	±0.19	**-0.06**	±0.26	0.23	±0.24	-	-	-	-	**-0.15**	±0.12	0.33	±0.25
Eclipse Core	**-0.08**	±0.23	0.12	±0.31	**-0.03**	±0.21	0.18	±0.37	**-0.03**	±0.23	0.15	±0.35	**-0.04**	±0.14	0.05	±0.14	**-0.08**	±0.16	0.44	±0.31
FreeMind	**-0.07**	±0.21	0.23	±0.41	0.01	±0.01	0.15	±0.36	0.01	±0.06	0.16	±0.32	-	-	-	-	**-0.05**	±0.07	0.14	±0.09
HSQLDB	**-0.07**	±0.21	0.15	±0.27	**-0.03**	±0.13	0.08	±0.14	**-0.04**	±0.12	0.14	±0.11	**-0.03**	±0.30	0.11	±0.15	**-0.09**	±0.22	0.38	±0.14
Apache Hive	**-0.04**	±0.16	0.14	±0.45	0.04	±0.14	0.12	±0.38	**-0.02**	±0.17	0.13	±0.43	**-0.01**	±0.10	0.03	±0.11	**-0.03**	±0.27	0.24	±0.36
Apache Ivy	**-0.10**	±0.23	0.19	±0.32	**-0.03**	±0.14	0.10	±0.37	**-0.01**	±0.18	0.10	±0.36	**-0.04**	±0.25	0.11	±0.09	**-0.02**	±0.14	0.27	±0.42
Apache Log4j	**-0.05**	±0.16	0.18	±0.13	0.01	±0.08	0.25	±0.12	**-0.01**	±0.13	0.21	±0.11	-	-	-	-	0.02	±0.11	0.32	±0.22
Apache Lucene	**-0.11**	±0.24	0.11	±0.32	**-0.02**	±0.07	0.12	±0.25	**-0.01**	±0.07	0.11	±0.23	0.01	±0.05	0.14	±0.06	**-0.03**	±0.07	0.38	±0.14
JEdit	**-0.11**	±0.25	0.12	±0.15	**-0.04**	±0.23	0.23	±0.35	**-0.02**	±0.25	0.23	±0.35	-	-	-	-	**-0.16**	±0.05	0.26	±0.11
JHotDraw	**-0.09**	±0.23	0.10	±0.27	**-0.04**	±0.23	0.13	±0.05	**-0.04**	±0.22	0.12	±0.19	**-0.07**	±0.10	0.01	±0.04	**-0.08**	±0.12	0.11	±0.24
JVLT	**-0.21**	±0.27	0.86	±0.03	**-0.10**	±0.41	0.68	±0.07	**-0.10**	±0.41	0.76	±0.20	-	-	-	-	**-0.04**	±0.17	0.08	±0.04
Apache Karaf	**-0.06**	±0.16	0.29	±0.12	**-0.02**	±0.15	0.16	±0.35	**-0.02**	±0.16	0.16	±0.38	**-0.07**	±0.34	0.05	±0.12	**-0.15**	±0.19	0.21	±0.34
Apache Nutch	**-0.09**	±0.24	0.12	±0.05	**-0.01**	±0.02	0.05	±0.32	**-0.02**	±0.08	0.05	±0.31	-	-	-	-	0.01	±0.27	0.23	±0.09
Apache Pig	**-0.02**	±0.30	0.09	±0.36	**-0.06**	±0.17	0.08	±0.31	**-0.01**	±0.12	0.11	±0.24	**-0.17**	±0.07	0.02	±0.13	**-0.24**	±0.25	0.12	±0.04
Apache Qpid	**-0.09**	±0.23	0.06	±0.41	**-0.08**	±0.29	0.11	±0.26	**-0.01**	±0.15	0.10	±0.22	**-0.23**	±0.17	0.25	±0.11	0.02	±0.13	0.04	±0.11
Apache Struts	**-0.04**	±0.14	0.10	±0.18	**-0.01**	±0.03	0.11	±0.27	**-0.02**	±0.11	0.10	±0.25	-	-	-	-	**-0.18**	±0.33	0.25	±0.12
Apache Xerces	**-0.06**	±0.19	0.16	±0.31	**-0.02**	±0.12	0.09	±0.31	**-0.03**	±0.12	0.10	±0.28	**-0.08**	±0.16	0.28	±0.29	**-0.03**	±0.20	0.29	±0.21
Overall	**-0.09**	**±0.24**	**0.14**	**±0.35**	**-0.03**	**±0.20**	**0.15**	**±0.35**	**-0.04**	**±0.19**	**0.14**	**±0.34**	**-0.06**	**±0.18**	**0.17**	**±0.22**	**-0.15**	**±0.23**	**0.37**	**±0.26**

metrics (*e.g.,* LCOM5) increase (worsen) by 86% on average at each new release. Instead, for the same project, the intensity of textual Blobs decreases (improves) 21% on average. An interesting example can be found in `Apache Ant`, when analyzing the evolution of the class `Property` of the `org.apache.tools.ant.taskdefs` package. The class is responsible for managing the Ant build properties. In the first versions of the project—from version 1.2 to version 1.5.4—the class was affected by a *Blob* code smell (it had a level of textual intensity equal to 0.83) since it implemented seven different ways to set such properties. During its evolution, the intensity has been reduced by developers through the application of different types of operations, such as code overriding (version 1.6) and refactorings (version 1.6.1), leading to a decrease of the complexity of the class, and consequently to the removal of the *Blob* smell. Currently, the class is responsible to set the execution environment of the build process by getting the desired properties using a string. A similar discussion can be made for the other studied code smells. Code elements affected by textual design flaws are seem-

Table 10.5: Comparison between Δ_{text} and Δ_{struct} for Blob, Feature Envy and Long Method. We use S, M, and L to indicate small, medium and large Cliff's d effect sizes respectively. Significant p-values are reported in bold face

	Textual vs. Structural														
	Blob			Feature Envy			Long Method			Misplaced Class			Promiscuous Package		
Project	**p-value**	d	**M**	**p-value**	d	**M**	**p-value**	d	**M**	**p-value**	d	**M**	**p-value**	d	**M**
ArgoUML	<**0.01**	-0.76	L	<**0.01**	-0.85	L	<**0.01**	-0.88	L	-	-	-	<**0.01**	-0.51	L
Apache Ant	<**0.01**	-0.66	L	<**0.01**	-0.67	L	<**0.01**	-0.71	L	<**0.01**	-0.79	L	<**0.01**	-0.62	L
aTunes	<**0.01**	-0.84	L	<**0.01**	-0.52	L	<**0.01**	-0.59	L	-	-	-	<**0.01**	-0.57	L
Apache Cassandra	<**0.01**	-0.76	L	<**0.01**	-0.89	L	<**0.01**	-0.91	L	-	-	-	<**0.01**	-0.60	L
Eclipse Core	<**0.01**	-0.83	L	<**0.01**	-0.77	L	<**0.01**	-0.74	L	<**0.01**	-0.76	L	<**0.01**	-0.78	L
FreeMind	<**0.01**	-0.78	L	<**0.01**	-0.83	L	<**0.01**	-0.79	L	-	-	-	<**0.01**	-0.72	L
HSQLDB	<**0.01**	-0.83	L	<**0.01**	-0.72	L	<**0.01**	-0.84	L	<**0.01**	-0.92	L	<**0.01**	-0.84	L
Apache Hive	<**0.01**	-0.65	L	<**0.01**	-0.70	L	<**0.01**	-0.68	L	<**0.01**	-0.89	L	<**0.01**	-0.98	L
Apache Ivy	<**0.01**	-0.89	L	<**0.01**	-0.78	L	<**0.01**	-0.63	L	<**0.01**	-0.73	L	<**0.01**	-0.66	L
Apache Log4j	<**0.01**	-0.76	L	<**0.01**	-0.82	L	<**0.01**	-0.78	L	-	-	-	<**0.01**	-0.79	L
Apache Lucene	<**0.01**	-0.83	L	<**0.01**	-0.89	L	<**0.01**	-0.91	L	<**0.01**	-0.68	L	<**0.01**	-0.71	L
JEdit	<**0.01**	-0.95	L	<**0.01**	-0.79	L	<**0.01**	-0.76	L	-	-	-	<**0.01**	-0.63	L
JHotDraw	<**0.01**	-0.80	L	<**0.01**	-0.92	L	<**0.01**	-0.89	L	<**0.01**	-0.72	L	<**0.01**	-0.61	L
JVLT	<**0.01**	-1.00	L	<**0.01**	-1.00	L	<**0.01**	-0.98	L	-	-	-	<**0.01**	-0.70	L
Apache Karaf	<**0.01**	-1.00	L	<**0.01**	-0.87	L	<**0.01**	-0.79	L	<**0.01**	-0.72	L	<**0.01**	-0.68	L
Apache Nutch	<**0.01**	-1.00	L	<**0.01**	-0.81	L	<**0.01**	-0.81	L	-	-	-	<**0.01**	-0.97	L
Apache Pig	<**0.01**	-0.78	L	<**0.01**	-0.90	L	<**0.01**	-0.82	L	<**0.01**	-0.64	L	<**0.01**	-0.71	L
Apache Qpid	<**0.01**	-0.70	L	<**0.01**	-0.84	L	<**0.01**	-0.85	L	<**0.01**	-0.59	L	<**0.01**	-0.65	L
Apache Struts	<**0.01**	-0.94	L	<**0.01**	-0.88	L	<**0.01**	-0.90	L	-	-	-	<**0.01**	-0.79	L
Apache Xerces	<**0.01**	-0.85	L	<**0.01**	-0.81	L	<**0.01**	-0.86	L	<**0.01**	-0.95	L	<**0.01**	-0.82	L
Overall	<**0.01**	**-0.78**	**L**	<**0.01**	**-0.78**	**L**	<**0.01**	**-0.77**	**L**	<**0.01**	**-0.74**	**L**	<**0.01**	**-0.69**	**L**

ingly more carefully managed by developers. On the other hand, code smells detected by DECOR tend to have a different evolution. For instance, the evolution of the method `org.hsqldb.JDBCBench.createDatabase` of the `HSQLDB` project is quite representative. This method should manage the functionality for creating a new database, but over history its size strongly increased as more sub-functionalities have been added, resulting in a *Long Method*. Interesting is the comment left by a developer in the source code of the method at version 1.7.3 of the project: *"Totally incomprehensible! One day or another, we should fix this method... I don't know how!"*. This comment gives strength to our initial conjecture, namely that source code characterized by inconsistent vocabulary is easier to perceive and, therefore, more urgent to maintain.

Our preliminary findings are also confirmed by the statistical tests, whose results are reported in Table 10.5. Specifically, for all the studied code smells the difference between the two distributions Δ_{text} and Δ_{struct} is always statistically significant (ρ-values$<$0.01), *i.e.*, the variations of intensity for structural and textual smells are statistically different. It is worth noting that the magnitude of Cliff's d measure is always *large*.

Having observed that textual and structural code smells are treated differently, we turn our attention to investigating which types of operations are performed by developers on the two sets of code smells, and to what extent such operations have an effect on the increase/decrease of their intensity. As for operations having the effect of increasing the intensity of textually and structurally detected code smells, we did not find a clear relationship between specific changes and the increase of intensity. Specifically, when considering textually detected smells, we found that for 35% of changes implementing new features the intensity tends to increase; 57% of times an increase is due to enhancement or bug fixing activities. Also for structurally detected smells, we observed that most of the times (91%) changes aimed at implementing new features, enhancing or fixing bugs in the project tend to increase the smell intensity. Moreover, for both textual and structural smells, we found that in a small percentage (8% for textual smells, 9% for structural smells) refactoring operations increase the level of intensity. Even though this result can appear unexpected, it confirms previous findings demonstrating that, rather than removing code smells, sometimes refactorings can be the cause of introducing design flaws [47].

Regarding the operations reducing the level of intensity, Table 10.6 reports the percentage of the different types of changes, *i.e.*, *New Feature* (NF), *Bug Fixing* (BF), *Refactoring* (R), and *Enhancement* (E), applied on the set of textual and structural code smells in our dataset. Considering the results achieved in previous work [107, 13, 40, 49], the most unexpected result is the one related to the percentage of refactoring operations. In fact, even if the number of refactoring operations performed on code smells remains quite low —confirming that code smells are poorly refactored— we observed that textual smells are generally more prone to

Table 10.6: Percentage of Different Types of Changes applied over Textual and Structural Code Smells. NF = New Feature; BF = Bug Fixing; R = Refactoring; E = Enhancement

Code Smell	Textual Smells				Structural Smells			
	NF	BF	R	E	NF	BF	R	E
Blob	10	32	14	44	10	32	10	48
Feature Envy	12	28	14	46	10	34	8	48
Long Method	8	34	13	45	8	38	6	48
Misplaced Class	15	21	14	50	17	25	5	53
Promiscuous Package	11	33	17	39	9	37	7	47

be subject to these operations (*Blob*=+4%, *Feature Envy*=+6%, *Long Method*=+7%, *Misplaced Class*=+9%, *Promiscuous Package*=+10%). In the following, we provide detailed results on the types of changes positively influencing the intensity for each code smell in our study.

Blob. Table 10.7 shows the Odds Ratios of the different types of changes applied on textual and structural code smells, obtained when building a logistic regression model for data concerning the decrease of smell intensity. In the following, we will mainly focus our discussion on statistically significant values. First of all, we can notice that changes tagged as *Refactoring* often have higher chance to decrease the intensity of textual *Blob*s (the ORs are higher than 1 in 85% of significant ORs). Thus, refactorings applied to *Blob* instances characterized by textual problems have a higher chance of being effective in the reduction of the smell intensity. Another unexpected result regards what we found for the category *Enhancement*: indeed, also in this case such changes have more chance to be effective in the reduction of the complexity of a textual *Blob*. A possible reason behind this finding concerns the higher ability of developers to enhance code components affected by textual smells, due to the lower difficulties they have to understand the problems affecting the source code. As for structural *Blob* instances, the results show that *Bug Fixing* operations have a higher chance to reduce the smell intensity. This means that code components having low quality as measured by software metrics are

Table 10.7: OR of different types of changes applied to *Blob*, *Feature Envy*, and *Long Method* instances when building logistic model. Statistically significant ORs are reported in **bold** face.

Project	Blob								Feature Envy								Long Method							
	Textual Smell				Structural Smell				Textual Smell				Structural Smell				Textual Smell				Structural Smell			
	NF	BF	R	E	NF	BF	R	E	NF	BF	R	E	NF	BF	R	E	NF	BF	R	E	NF	BF	R	E
ArgoUML	0.78	0.85	1.11	**1.50**	0.81	**1.00**	0.87	0.89	0.81	0.97	0.89	**1.10**	0.80	**0.83**	**0.93**	0.95	0.87	**1.05**	0.99	1.01	0.73	0.89	**0.89**	0.74
Apache Ant	0.99	**1.01**	1.00	1.01	0.97	1.01	**0.99**	**1.00**	0.88	**0.93**	0.91	**1.02**	0.87	**0.82**	0.81	**0.92**	0.86	1.01	**1.11**	**1.02**	0.69	**0.98**	0.84	0.89
aTunes	1.01	0.98	**1.01**	**1.14**	0.98	**1.01**	0.95	0.87	**0.98**	0.89	**0.92**	1.01	0.87	**0.77**	0.91	0.96	0.89	**1.02**	**1.10**	1.00	0.99	0.97	**1.01**	0.88
Apache Cassandra	0.99	1.00	1.00	1.00	0.97	**1.01**	0.99	1.00	0.83	**0.87**	0.92	**1.03**	0.87	**0.78**	**0.76**	0.93	0.93	**0.98**	**1.01**	1.01	0.89	**0.92**	**0.95**	0.91
Eclipse Core	**0.88**	1.01	**1.10**	**1.34**	0.99	**0.97**	1.00	**1.02**	0.81	0.84	**0.97**	**1.04**	0.83	0.92	0.83	**0.97**	0.98	**1.10**	**1.15**	0.97	0.68	**1.02**	0.98	**1.01**
FreeMind	0.91	**1.02**	**0.89**	**1.22**	0.92	**0.98**	0.78	0.88	0.72	0.81	0.94	**1.02**	0.71	**0.82**	**0.91**	0.99	**0.86**	**1.02**	0.99	0.87	0.83	**0.97**	0.94	0.91
HSQLDB	1.01	0.97	**1.06**	**1.18**	1.01	0.93	**1.00**	0.99	0.75	**0.81**	0.98	1.10	**0.72**	**0.89**	0.93	0.84	0.92	0.97	**1.02**	**0.88**	0.91	0.97	**0.99**	**0.78**
Apache Hive	0.99	1.00	**1.01**	1.00	0.97	**1.01**	0.99	1.00	0.87	**0.93**	**0.84**	1.11	0.73	**0.74**	0.81	0.96	**0.89**	0.97	**1.00**	0.87	0.79	**0.81**	0.74	**0.89**
Apache Ivy	0.99	1.00	**1.02**	1.00	0.96	**0.99**	**0.99**	1.00	0.74	0.73	0.92	**1.08**	0.86	0.74	**0.72**	0.83	0.92	0.98	**1.01**	0.98	0.91	0.86	0.81	**0.92**
Apache Log4j	0.86	**1.02**	**1.06**	**1.19**	0.94	**0.98**	**1.01**	**1.01**	**0.83**	**0.92**	0.97	1.05	0.78	0.70	**0.91**	0.96	0.91	**1.02**	**1.10**	0.82	**0.72**	0.76	**0.93**	0.83
Apache Lucene	0.99	1.00	**1.20**	1.00	0.97	**1.01**	0.99	1.00	0.81	0.84	0.92	**1.04**	0.79	**0.84**	0.81	0.97	0.97	**1.14**	**1.36**	**1.02**	0.89	0.93	**1.05**	**0.82**
JEdit	0.88	1.04	1.18	**1.24**	0.0	0.65	0.18	0.66	**0.83**	0.98	0.82	1.03	0.85	**0.78**	0.97	0.98	**0.85**	**1.01**	0.99	0.84	0.92	**0.78**	0.92	0.89
JHotDraw	0.67	0.88	**1.02**	**1.09**	0.54	0.41	**0.88**	0.64	0.84	**0.87**	0.93	**0.99**	0.84	0.78	0.84	**0.91**	0.92	**1.21**	0.97	0.88	**0.93**	**0.99**	0.96	0.88
JVLT	**0.51**	0.75	**1.01**	0.97	0.87	**0.99**	**1.01**	0.77	0.95	0.83	0.99	**1.06**	0.91	0.84	**0.93**	0.97	0.88	1.02	**1.02**	0.87	**1.00**	0.99	0.99	0.97
Apache Karaf	0.99	1.00	1.00	**1.01**	0.96	1.00	**0.99**	1.00	0.86	**0.88**	0.98	**1.11**	0.87	**0.94**	0.82	0.98	0.91	1.01	0.97	**0.89**	0.96	**1.01**	0.89	**0.79**
Apache Nutch	0.99	1.00	**1.02**	1.00	**0.95**	0.99	0.93	1.00	0.81	0.78	**0.98**	1.08	0.89	**0.79**	**0.83**	0.93	0.86	0.97	1.03	**1.12**	**0.78**	**0.85**	0.96	0.98
Apache Pig	0.99	1.01	**1.12**	**0.97**	0.96	**1.01**	0.99	1.00	0.88	0.87	0.92	**1.07**	0.71	**0.76**	**0.77**	0.86	0.93	0.99	**1.02**	**1.12**	**0.87**	0.79	**0.96**	1.02
Apache Qpid	0.99	**1.10**	**1.04**	1.00	0.97	0.99	**1.00**	**1.01**	0.87	0.82	0.88	**1.12**	0.72	**0.86**	0.84	0.91	0.82	1.00	1.01	**1.22**	0.87	**1.02**	0.98	0.92
Apache Struts	0.87	**1.05**	**1.15**	**1.27**	0.93	0.97	0.91	**0.89**	0.87	0.97	**1.02**	**1.17**	0.73	0.83	**0.98**	**0.98**	0.76	**1.01**	1.06	1.11	0.87	0.99	**0.98**	**1.02**
Apache Xerces	0.92	1.01	**1.52**	**1.23**	0.95	**0.97**	0.91	**0.99**	0.78	0.92	0.97	**1.21**	0.86	0.83	0.92	**1.02**	1.01	**1.09**	1.05	**1.12**	**0.98**	**1.01**	0.99	1.03
Overall	**0.99**	**1.02**	**1.23**	**1.20**	**0.97**	**1.01**	**0.98**	**0.99**	**0.81**	**0.85**	**0.91**	**1.14**	**0.79**	**0.86**	**0.91**	**0.93**	**0.96**	**1.05**	**1.18**	**1.02**	**0.88**	**0.95**	**0.97**	**0.97**

mainly touched by developers only when a bug fix is required. Looking at these findings, *we can conclude that textual Blob instances are on the one hand more prone to be refactored and, on the other hand more likely to be resolved by such operations, while the complexity of structural Blob instances is mainly reduced through bug fixing operations*. This claim is also supported by the analysis of the number of textual and structural *Blob* instances actually removed in our dataset (see Table 10.8). Indeed, we observed that 27% of textual smells in our dataset have been removed over time, and in 12% of the times they have been removed using refactoring operations. On the other hand, we can see that the percentage of structural *Blob* instances removed over time is much lower (16%), and the percentage of refactorings is 7% lower with respect to textual blobs.

Feature Envy. The ORs achieved when applying the logistic regression model relating the types of changes to the decrease of smell intensity for *Feature Envy* are reported in Table 10.7. In this case, changes classified as *New Feature*, *Bug Fix-*

Table 10.8: Percentage of Removed Textual (TS) and Structural Code (SS) Smell instances

Code Smell	% TS Removed (% due to refact.)	% SS Removed (% due to refactor.)	Residual
Blob	27 (12)	16 (5)	**+11 (+7)**
Feature Envy	16 (4)	11 (2)	**+5 (+2)**
Long Method	35 (18)	20 (8)	**+15 (+10)**
Misplaced Class	18 (11)	7 (3)	**+11 (+8)**
Promiscuous Package	26 (11)	13 (4)	**+13 (+7)**

ing, and *Refactoring* in most cases do not reduce the intensity of either textual and structural *Feature Envy* instances. Instead, the enhancement operations made on textually detected *Feature Envy* smells have, overall, 14% more chance of reducing the smell intensity. Looking at the results of structurally detected *Feature Envy*, none of the analyzed changes seem to lead to an intensity reduction. Moreover, it seems that textually detected *Feature Envy* instances differ from structurally detected ones, since other than being removed more frequently (see Table 10.8), the refactoring operations are slightly more effective (+2%) in the removal of the smell. Since this smell arises when a method has more in common with another class with respect to the one it is actually in, such a difference can be explained considering the way developers perceive different types of coupling. Indeed, as shown by Bavota *et al.* [180], conceptual coupling better reflects the mental model of developers. So, it seems that developers are able to better perceive the symptoms of a textual *Feature Envy*, by providing solutions to limit it, or in some cases provide accurate solutions to remove it.

Long Method. Table 10.7 reports the results achieved when comparing textually and structurally detected *Long Method* instances. Regarding textual long methods, there are several causes which relate with the decrease of their intensity. Overall, we can see that refactorings have 18% more chance of reducing the complexity of the smell, while *Bug Fixing* and *Enhancement* operations have 5% and 2% more chance, respectively. These findings highlight how the decrease of the intensity

Table 10.9: OR of different types of changes applied to *Misplaced Class* and *Promiscuous Package* instances when building logistic model. Statistically significant ORs are reported in **bold** face.

Project	Misplaced Class								Promiscuous Package							
	Textual Smell				Structural Smell				Textual Smell				Structural Smell			
	NF	BF	R	E	NF	BF	R	E	NF	BF	R	E	NF	BF	R	E
ArgoUML	-	-	-	-	-	-	-	-	1.02	0.93	**1.04**	0.99	0.97	0.98	0.88	1.01
Apache Ant	0.81	**1.04**	**1.08**	0.91	0.82	1.01	0.79	**1.01**	0.96	0.80	**1.02**	0.71	**1.02**	0.82	0.94	**1.01**
aTunes	-	-	-	-	-	-	-	-	0.82	**1.01**	0.77	0.92	**1.01**	0.67	0.92	0.81
Apache Cassandra	-	-	-	-	-	-	-	-	0.72	1.04	**1.11**	0.77	0.80	**1.01**	0.99	0.60
Eclipse Core	0.97	1.01	**1.18**	0.83	0.98	0.88	0.91	1.01	1.01	0.92	**1.12**	0.84	0.95	**1.08**	0.81	0.72
FreeMind	-	-	-	-	-	-	-	-	0.84	1.02	0.97	0.99	0.98	**1.01**	0.85	0.68
HSQLDB	**0.86**	0.79	**1.17**	0.80	0.71	0.66	0.91	**1.02**	0.51	**1.01**	**1.02**	0.81	0.98	**1.05**	0.95	0.83
Apache Hive	0.91	0.87	**1.18**	0.98	0.79	0.71	0.92	**1.02**	0.89	0.99	0.80	0.77	0.92	0.82	0.84	**1.02**
Apache Ivy	0.86	0.78	0.99	0.92	0.78	0.92	0.87	**1.06**	0.88	0.61	**1.11**	0.87	0.91	0.82	0.65	**1.03**
Apache Log4j	-	-	-	-	-	-	-	-	0.72	0.69	**1.20**	0.92	0.55	**1.06**	0.91	0.83
Apache Lucene	0.81	1.04	**1.21**	0.72	0.52	0.64	0.89	**1.01**	**1.03**	0.98	**1.02**	0.78	0.56	0.61	0.54	**0.89**
JEdit	-	-	-	-	-	-	-	-	0.88	**0.99**	0.92	0.88	0.79	0.76	0.66	**0.93**
JHotDraw	0.82	1.01	**1.22**	0.91	0.88	0.87	0.72	**0.84**	0.72	0.77	1.05	1.08	**0.79**	0.82	0.99	0.60
JVLT	0.51	-	-	-	-	-	-	-	0.95	0.73	**1.09**	0.74	0.69	0.59	0.68	0.99
Apache Karaf	0.81	**1.01**	0.90	0.82	0.81	0.57	0.82	**0.95**	0.69	0.72	**1.01**	0.92	0.74	0.92	0.81	**1.04**
Apache Nutch	-	-	-	-	-	-	-	-	0.77	**1.02**	**1.08**	0.98	0.96	0.88	**0.92**	0.81
Apache Pig	1.01	0.81	**1.27**	0.89	0.52	0.76	0.99	**0.92**	0.78	0.91	**1.21**	0.72	0.74	0.72	0.89	0.71
Apache Qpid	0.83	0.88	**1.11**	0.98	1.01	**1.05**	0.66	0.82	0.60	0.72	**1.20**	0.82	0.72	0.61	0.86	0.87
Apache Struts	-	-	-	-	-	-	-	-	0.81	**1.01**	**1.06**	0.62	0.81	0.86	**0.99**	**1.02**
Apache Xerces	0.92	0.99	**1.09**	**1.02**	0.82	**1.01**	0.89	**1.05**	0.99	**1.05**	**1.14**	0.74	0.88	**1.04**	0.89	0.71
Overall	**0.89**	**0.99**	**1.15**	**0.92**	**0.76**	**0.91**	**0.94**	**1.02**	**0.86**	**1.01**	**1.10**	**0.84**	**0.93**	**0.98**	**0.85**	**0.98**

of textual long methods depends on the activity of the developers. When analyzing the results for structurally detected long methods, we observed that there are no specific types of changes that strongly influence the decrease of smell intensity. Thus, looking at these results, also in this case we can affirm that textual long methods have characteristics which help developers in detecting and, thus, removing them from the source code. Table 10.8 highlights these differences, since textually detected *Long Method* instances are removed in 35% of cases over time (18% of removals are due to refactoring), while structurally detected long methods are removed in 20% of the cases (8% of cases due to refactoring activities).

Misplaced Class. When evaluating the differences between textually and structurally detected instances of this smell (Table 10.9), we can delineate a clear trend in the results. Indeed, textual *Misplaced Class* instances undergo a considerable reduction of their intensity when refactoring is applied (changes of this type have 15% more chance of reducing the intensity of the smell). The claim is also supported by the percentage of smell instances removed by developers shown in Table 10.8, where we can observe that the smell is removed in 18% of the cases over time and, more importantly, 11% of removals are due to refactoring. As previously explained for the *Feature Envy* smell, the reasons behind this strong result can be found in the developers' perception of software coupling [180]. In fact, the smell arises when a class has responsibilities closer to the ones of another package with respect to the one it is actually in. Therefore, if developers better comprehend the conceptual relationships between classes it is reasonable to assume that they are more inclined to move classes to better locations. Concerning instances of the smell found by the structural approach, the situation is different because of the limited values of the ORs achieved by the considered types of changes. The higher value is the one found by considering changes of the category *Enhancement* (+2% more chance of reducing the intensity). This means that developers actually reduce the intensity of structural misplaced classes only in cases where an enhancement is needed, rather than limiting the intensity through appropriate refactoring operations. It is worth noting that the percentage of instances of this smell removed over time is lower than the one of textual smells (-11% of removals) and that refactoring is the cause of removal only in 3% of the cases.

Promiscuous Package. The discussion of the results for this smell type is quite similar to the ones above. Indeed, from Table 10.9 we can observe that in most cases refactorings are limiting the intensity of textually detected instances (+10% chance to reduce the severity), while for structural smells there are no specific types of changes that influence the decrease of smell intensity. This result seems to highlight the higher ability of developers to deal with promiscuous packages characterized by scattered concepts rather than by structural symptoms. There-

fore, also in this case we can affirm that the characteristics of textually detected instances of this smell help developers in finding adequate solutions to reduce the intensity of the design flaw. Notably, such instances are removed in 26% of the cases over the release history of the projects analyzed (+13% with respect to structural instances), and 11% of the times the reason of the removal is refactoring (+7% than structural promiscuous packages).

10.5.3 Threats to Validity

Threats to *construct* validity concern the relationship between theory and observation, and are mainly related to the measurements we performed in our study. Specifically, we monitored the evolution of the level of smelliness of textual and structural code smells by relying on five tools, *i.e.,* TACO, DECOR [20], *JDeodorant* [25], and the approaches by Girvan *et al.* [196] and Atkinson and King [76], empirically validated and providing good performance in code smell detection. Nevertheless, we are aware that our results can be affected by the presence of false positives and false negatives. While we re-implemented all the structural-based techniques but *JDeodorant*, we followed the exact algorithms defined by the corresponding authors. Moreover, to make sure that our implementation was not biased, we replicated the studies presented in [20, 76, 196], obtaining similar results. For the *commit goal* tag assignment to commits involving code smells, we used methodologies successfully used in previous work [47]. Moreover, we manually validated all the tags assigned to such commits.

Threats to *internal* validity concern factors that could have influenced our results. The fact that design flaws are removed may or may not be related to the types of changes we considered in the study, *i.e.,* other changes could have produced such effects. Since the findings reported so far allow us to claim correlation and not causation, in Section 10.6 we corroborate our quantitative results with a user study where we involve industrial developers and software quality experts, with the aim of finding a practical explanation of our quantitative results.

Threats to *external validity* concern the generalization of the results. A specific

threat to external validity is be related to the number of subject systems used in our empirical evaluation. To show the generalizability of our results, we conducted an empirical study involving 20 Java open source systems having different size and different domains. However, it could be worthwhile to replicate the empirical study on other projects written in different programming languages.

10.6 Study II: The Perception of Textual and Structural Code Smells

In the previous section we found evident differences in the way developers treat textual and structural smells. However, we cannot speculate on the reasons behind such differences by solely looking at historical data. Indeed, on the one hand it is possible that developers perceive textual smells more as design problems than structural smells, while on the other hand it is also possible that this difference relies on the fact that textual smells are easier to maintain. To better understand the reasons behind the results achieved in the previous analysis, we conducted a qualitative study aimed at investigating how developers perceive structural and textual instances of smells.

10.6.1 Empirical Study Definition and Design

This study reports on a qualitative investigation conducted with (i) professional developers through a questionnaire, and (ii) software quality experts through semi-structured interviews, with the *goal* of investigating their perception of code smells with respect to different sources of information, *i.e.,* textual and structural. Hence, we designed this study to answer the following research questions (**RQs**):

> **RQ3:** *Do developers perceive design problems behind textual smells more than design problems behind structural smells?*
>
> **RQ4:** *Do developers find textual smells easier to refactor than structural smells?*

The *context* of the study consists of two software projects, *i.e.,* Eclipse 3.6.1 and Lucene 3.6 that have already been used in Section 10.5. For each of them, we selected four instances of *Blob*, *Feature Envy*, *Long Method*, *Misplaced Class*, and *Promiscuous Package* adhering to the following process:

1. we computed the level of structural and textual intensity for all packages, classes, and methods in Eclipse and Lucene;

2. we selected two instances of each code smell type correctly detected by structural tools but not by TACO;

3. we also selected two other instances of each type of code smell that are correctly classified by TACO and not by structural tools.

Note that we selected instances classified by the detectors as the ones having highest intensity (*i.e.,* the selected instances have similar intensity values). Moreover, to avoid possible biases caused by the interaction of more code smells [9, 18], we selected instances affected by only a single type of code smell. Finally, we also randomly selected two code elements of different granularity (*i.e.,* a package, a class, or a method) not affected by any of the code smells considered in our study. This was done to limit the bias in the study, *i.e.,* avoid that participants always indicated that the code contained a problem and the problem was a serious one.

To answer our research questions, we invited industrial developers from different application domains having a programming experience ranging between 2 and 7 years and that generally work on Java development. The invitations were sent via e-mail. We have contacted 60 developers in total, receiving 10 responses. Each one performed the tasks related to a single software systems.

Participants received the experimental material via the `eSurveyPro`[5] online platform and, hence, they used their own personal computer with their preferred IDE to answer the proposed questions. The survey provided (i) a pre-test questionnaire, (ii) detailed instructions to perform the experiment, and (iii) the source code of the two projects involved in the study, *i.e.,* Eclipse and Lucene. During

[5] `http://www.esurveyspro.com`

the initial step, participants were asked to sign a statement of consent and fill in a pre-test questionnaire in which we collected information about their programming experience and background on code smells. Afterwards, each participant was required to inspect a total of twelve code elements related to one of the two projects in our study, namely five pairs of instances affected by either structural or textual code smells (*i.e., Blob*, *Feature Envy*, *Long Method*, *Misplaced Class*, and *Promiscuous Package*) plus a pair of code elements not affected by design flaws.

We balanced the 10 participants so that five participants analyzed code smell instances for Eclipse, while the other five participants focused on Lucene.

The experiment was composed of six consecutive sessions with two tasks each. During each session, participants performed two tasks related to the same type of code smell but detected by different tools: one task involving a code smell detected by TACO and a second task related to the same type of code smell but detected by structural-based tools. In each task, participants were asked (i) to analyze the target code component (either a package, a class or a method), (ii) to fill in a post-task questionnaire indicating whether the code component is affected by a design flaw or not, and (iii) to suggest (if a design flaw is detected) possible refactoring operations aimed at removing the smell. Moreover, participants were also asked to evaluate the proneness of the code element analyzed to be involved in a design flaw as well as the severity of different source code properties (*e.g.,* size, complexity, number of dependencies, etc.) using a Likert scale ranging from 1 to indicate a very low risk to 5 to denote very high risk. Participants were also allowed to browse other classes or methods to better understand the responsibilities of the code component under analysis and find possible dependencies. Clearly, we did not reveal the types of code smells, nor whether they were detected by structural or textual tools.

Participants were instructed to spend no more than 30 minutes for completing each task and they were allowed to finish earlier if and only if they believed that all design flaws were found (if any) *and* the corresponding refactoring operations were identified. Participants had up to four weeks to complete the survey.

To complement the analysis and receive opinions from people having a solid

knowledge about source code quality and code smells, in this stage we also recruited five software quality consultants: four from the `Software Improvement Group` (SIG) in the Netherlands[6], while the remaining one from the `Continuous Quality in Software Engineering` (CQSE) in Germany[7]. Both companies carry out software quality assessments for their customers. All participants have an average industrial experience of 4 years, an average programming experience of 9 years (one of them 20 years). This experiment has been conducted in the headquarters of the SIG company when we interviewed the consultants from SIG, while the quality expert from CQSE has been interviewed via Skype.

The consultants were asked to fill-in the same questionnaire provided to the industrial developers. However, due to time constraints, in this case the quality experts only answered the questions related to a subset of code smells, *i.e.*, *Blob*, *Feature Envy*, and *Long Method*. Also in this case, we distributed the experimental material in order to have a balanced number of answers between the considered software systems. For this reason, three experts answered questions related to Eclipse, the other two worked on code elements from Lucene.

At the end of the experiment, the quality experts were also required to participate in an open discussion session of 30 minutes to reflect on the tasks performed and to answer questions that we used to collect feedback for qualitative analysis. In particular, two of the authors[8] first asked them to walk through the classes in order to explain the responsibilities they implemented; then, we asked them to explain if and how they have identified a design problem in the source code (*e.g.*, which characteristic of the source code allowed them to recognize a design flaw). The discussion session was conducted by the first two authors of the corresponding paper. The total duration of the experiment was 2 hours and 30 minutes, including the time needed to complete the experimental sessions, to fill in the questionnaires and participate in the discussion.

The data collected by the post-task questionnaires are used to answer **RQ3** and

[6] `https://www.sig.eu/en/`

[7] `https://www.cqse.eu/en/`

[8] The thesis' author was equally involved in this task.

RQ4. Specifically, we address **RQ3** by analyzing the design flaws identified by each participant for the code components; we compared their classification to the classification made by the textual and structural tools (*i.e., Blob*, *Feature Envy*, *Long Method*, *Misplaced Class*, and *Promiscuous Package*). Moreover, we also compare the distributions of the Likert values assigned by participants when providing the indication of the proneness of a code element to be involved in a design flaw. In this way, we measure the perceived levels of risk for the two groups {*textual, structural*} of code smells to investigate how participants perceive the strength of code smells as identified by textual and structural based tools. To verify the statistical significance of the differences between the two distributions (*i.e.,* risk levels perceived for textual and structural smells) we use the non-parametric Wilcoxon Rank Sum test with a significance threshold of $\rho - value = 0.05$.

For **RQ4**, we analyze the refactoring operations suggested by participants for removing the identified code smells. For each different type of code smell there is, indeed, a specific set of refactoring operations defined suitable to address the design problems. The refactoring associated with *Long Method* is *Extract Method*, for *Feature Envy* the corresponding refactoring is *Move Method*, for *Blob* an *Extract Class* refactoring is advised, for *Misplaced Class* a *Move Class* refactoring should be applied, and for *Promiscuous Package* the associated solutions is represented by the *Extract Package* refactoring [8].

Finally, we provide an overview of the discussion, as well as hints provided by quality consultants during the open discussion session that followed their experiment.

10.6.2 Analysis of the Results

Before answering the two research questions formulated in the previous section, we analyze to what extent professional developers perceived the presence of a design problem in code elements not containing any of the smells considered in our study. As previously explained, this is a sanity check aimed at verifying whether participants were negatively biased. As a result, none of the involved developers

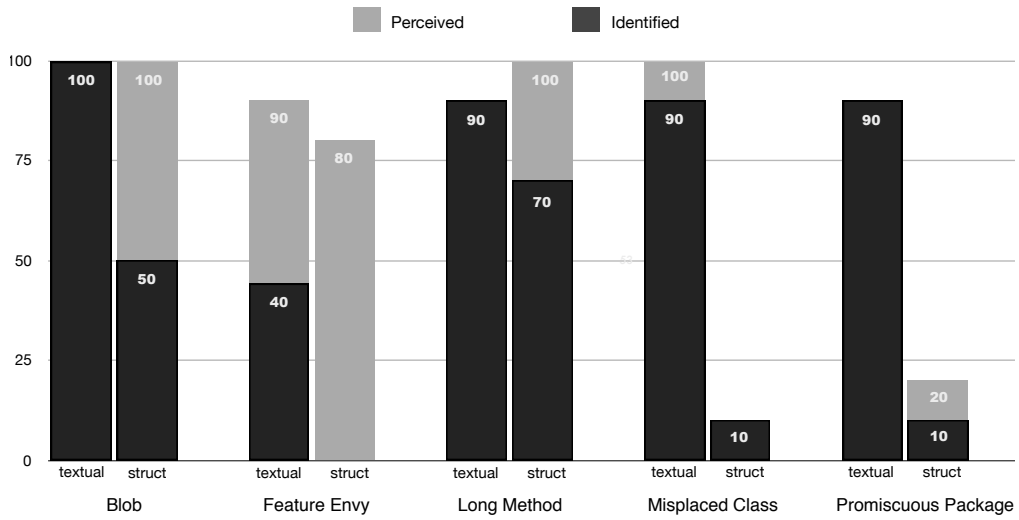


Figure 10.1: Percentage of Textual and Structural Code Smells Perceived and Identified by Professional Developers.

marked such code elements as affected by code smells. Hence, this result indicates the absence of a negative bias in the respondents.

Turning to the answers provided by participants when analyzing code elements affected by a code smell, Figure 10.1 depicts the bar plots reporting the percentage of code smells perceived (in grey) and identified (in black) by the professional developers who participated to the survey. Note that a code component that is correctly identified is also perceived (the opposite is not true). Moreover, Table 10.10 reports the percentage of refactoring operations correctly suggested by participants. Finally, Table 10.11 illustrates the design flaws observed as well as the refactoring operations suggested by each quality expert involved in the study. The rows represent participants while the columns depict the type of code smell (*i.e.*, Blob, Feature Envy and Long Method) and the source of information used for the detection (*i.e.*, structural or textual).

Looking at the Figure 10.1 we can immediately make two important general observations: first of all, in most cases textual smells are considered actual problems by developers, even though they do not exceed any structural metrics' value. Secondly, not only the analysis reveals important differences between textual and

Table 10.10: Percentage of refactoring operations correctly suggested by professional developers.

Code Smell	Structural		Textual	
	#	%	#	%
Long Method	7	70%	8	80%
Feature Envy	0	0%	2	20%
Blob	4	40%	8	80%
Promiscuous Package	1	10%	9	90%
Misplaced Class	1	10%	9	90%

structural design flaws in the way developers perceive them, but also that textually detected smells are generally correctly identified (*i.e.,* the description of the design problem matches the definition provided by Fowler [8]). At the same time, professional developers are also generally able to provide good suggestions on how to refactor the code smells that they correctly identify (see Table 10.10). This is particularly true for textual smells, where the developers almost always indicate the right refactoring to apply (for all the smells but *Feature Envy* more than 80% of participants described well the refactoring aimed at removing the design flaw). This trend is confirmed when considering the answers provided by software quality consultants (see Table 10.11). In this case, we can observe that in all the cases the quality experts perceived design flaws in the proposed code components.

Blob. Blob instances characterized by textual problems are always perceived and correctly classified by the participants. On the other hand, only half of the structurally detected instances are recognized by developers, even if the problem has always been perceived. While this result is in line with previous findings demonstrating that complex or long code is more easily detectable by developers [50], the different nature of the smells lead us to think that the textual properties of source code help developers in better understanding the actual flaws.

As for the quality consultants, structural blobs are correctly identified by par-

Table 10.11: Results obtained from Study II. Labels marked with (*) indicate that the design flaws identified by participants match the classification provided by textual or structural tools

Participant	Structural						Textual					
	Blob		Feature Envy		Long Method		Blob		Feature Envy		Long Method	
	Flaws	Refactoring	Flaws	Refactoring	Flaws	Refactoring	Flaws	Refactoring	Flaws	Refactoring	Flaws	Refactoring
Expert1	Blob*	Extr. Class*	High Complex.		Long Method*		Blob*	Extr. Class*	Feature Envy*	Move Method*	Long Method*	Extr. Method*
	Long Param. List						Long Param. List		High Complex.			
							High Complex.					
Expert2	Code Duplic.	Clone Reun.	High Complex.	Rename	Long Method*	Extr. Method*	Blob*	Extr. Class*	High Complex.	Rename	Long Method*	Extr. Method*
									Bad Identifiers		High Complex	
Expert3	Blob*	Extr. Class*	Long Method	Extr. Method	Long Method*	Extr. Method*	Blob*	Extr. Class*	Long Method	Extr. Method	Long Method*	Extr. Method*
							Bad Identifiers				Feature Envy	Move Method
							Redundant Code					
Expert4	Complex Class		Long Method	Extr. Method	Long Method*	Introd. Polym.	Blob*	Extr. Class*	Feature Envy*	Move Method*	Long Method*	Extr. Method*
				Extr. Class							Bad Identifiers	
Expert5	Blob*	Extr. Class*	Long Method	Extr. Method	Long Method*	Extr. Method*	Blob*	Extr. Class*	Long Method	Extr. Method	Long Method*	Extr. Method*
									Bad Identifiers	Rename		

ticipants in only three out of five cases. Note that in these cases participants also correctly indicate the refactoring operation (*i.e., Extract Class*) for removing the smell. In the remaining two classes, they describe symptoms referable to other code smells, such as *Complex Class* [61] and *Duplicate Code* [8]. Moreover, participants have not been able to identify an appropriate refactoring to remove these design flaws. During the open discussion, the quality experts were invited to think aloud [209] on the design problems affecting the analyzed code components. In this session, participants re-elaborated their analysis, correctly detecting the previously missed *Blob* instances. They also explained that the main reason of the misclassification is *the extreme complexity of the* (structurally detected) *code components that does not allow the correct identification of the design flaws affecting them*. For textual blobs, the discussion is different. Here we observed a complete agreement with the experts' perception: all classes detected as blobs by TACO are also identified as actual *Blob* instances by the participants. At the same time, it is worth observing that participants also identified the correct refactoring operation aimed at removing the smell.

The open discussion session confirms our conjecture, *i.e.,* the textual component of the source code actually help developers in understanding the design problems affecting a class.

Long Method. When considering the *Long Method* smell, the discussion is quite similar. Indeed, here we found that almost all the industrial developers perceived a problem, still confirming previous findings in the field [50]. However, there are two details to further analyze. In the first place, unlike the structurally detected ones all the long methods detected by TACO and perceived by developers have also been correctly identified. On the other hand, there was a specific case in which a textual *Long Method* instance has been marked as not affected by design flaw. It is the `ASTParser.createASTs` method of the Eclipse project, which is responsible for the analysis of the source code classes of a Java project and the subsequent creation of the Abstract Syntax Tree (AST) for all of them. Even if the method does not have an excessive length (*i.e.,* 68 lines of code), it should be considered as a design problem since it manages all the operations needed to build the AST of a class (*i.e.,* reading, parsing, and AST building). However, one of the professional developers involved in the study did not perceive this code smell as such.

On the other hand, the quality experts perceived and correctly identified all instances of long methods detected by either approach. However, Table 10.11 highlights evident differences between the refactoring operations suggested by participants to remove structural and textual smells. Indeed, while for structurally detected code smells the correct refactoring (*i.e., Extract Method*) has been identified in only three out of five cases, for textual smell participants always indicate the *Extract Method* refactoring as ideal solution for removing the identified smells. For instance, when indicating the possible refactoring for the long method `TieredMergePolicy.findMerges` of the `Apache Lucene` project, a quality consultant answered that "*parts of the method are separated by comments; at which points usually an extract method refactoring should be applied; then the names of the new methods can replace the comments*". This example is quite representative since the quality expert not only identified the correct refactoring to use, but also gave us hints on how the refactoring may be applied practically (*i.e.,* by splitting the method being driven by the comments that separate the different portions of code to extract).

Feature Envy. For the *Feature Envy* we obtain different results as compared to the two type of code smell discussed above. While the problem is generally perceived by professional developers, none of them was able to characterize the symptoms behind the structural instances of this smell. Conversely, textual instances are perceived more (90% vs 80%) and in some cases (40%) participants were able to describe the problem well. As for the refactoring operations suggested, only two developers who identified the problem correctly suggest the application of a *Move Method* refactoring. For instance, let us consider the case of the `getFieldQuery` method, contained in the `QueryParser` class of the Lucene system, affected by *Feature Envy* because it should be placed in the `PhraseQuery` class to which it is more closely related. When analyzing it, developer #6 claimed that "*the method seems to be more related to the class PhraseQuery, even if there are not so many dependencies between the two classes*".

The observations that we collect from surveying the quality experts are in line with those reported above. Indeed, in the majority of the cases the consultants did not classify this type of code smell correctly, even if they perceived that the subject code components actually had some design problems. Also in this case, the result is in line with previous research [50]. However, it is worth noting that the method `getMatchRuleString` from the class `BasicSearchEngine` (Eclipse), and the method `explain` from the class of Apache Lucene `DisjunctionMatchQuery` have been correctly classified as *Feature Envy* instances in agreement with the detection made by TACO (textual tool), and, at the same time, correct refactoring solutions have been suggested (*Move Method* refactoring). On the one hand, the higher ability of developers to detect textual *Feature Envy* instances is a likely consequence of the fact that conceptual coupling is more easily perceived than structural coupling [180]. On the other hand, hints provided by participants in the open discussion can allow us to draw a conclusion on why developers often do not identify this type of smell. Indeed, they reported that "*dependency on other classes is the less important point*", indeed, "*as long as a method is short and well documented (e.g., with proper identifiers), there is no real need to move it to a different class*".

Misplaced Class and Promiscuous Package. The importance of textual aspects of source code for the practical usefulness of code smell detectors is even more evident when considering *Misplaced Class* and *Promiscuous Package* smells. Indeed, in these cases the structurally detected instances are almost never perceived and identified, while the results achieved for textual instances are exactly the opposite (see Figure 10.1). At the same time, it is worth observing that the higher the granularity of the code smell, the lower the ability of the developers in perceiving structurally detected code smells. A possible reason behind this result is related to the need of developers of having higher level information to build a cognitive model to comprehend larger parts of source code [210, 211]. In this sense, the textual component can give a strong contribution to comprehend the problems affecting a higher-level code component such as a package.

General observations. In general, we observe two different trends for the type of design flaws (Table 10.11) detected by quality experts depending on whether code components are affected by structural and textual smells. While for structurally detected code smells (*i.e.,* by DECOR or JDeodorant) the experts generally identified only one design flaw per code component (often missing the actual problem affecting the code analyzed), for textual smells participants detected, other than the main one, also other problems related to textual aspects of source code, such as the presence of poor identifiers. For example, *Expert3* identified three different design flaws for the class `OperatorExpression` extracted from the Eclipse project: it is a *Blob* (in agreement with TACO) affected by redundant code and with meaningless identifiers.

At the end of each task, participants were asked to evaluate the severity of different properties (*e.g.,* size, complexity, number of dependencies, etc.) for the analyzed code using a Likert scale intensity from very-low to very-high. Figure 10.2 compares the scores assigned to the code smells according to whether they were detected by textual or structural tools. Such an analysis allows us to gain more insight into the perception of code smells stemming from different sources of information.

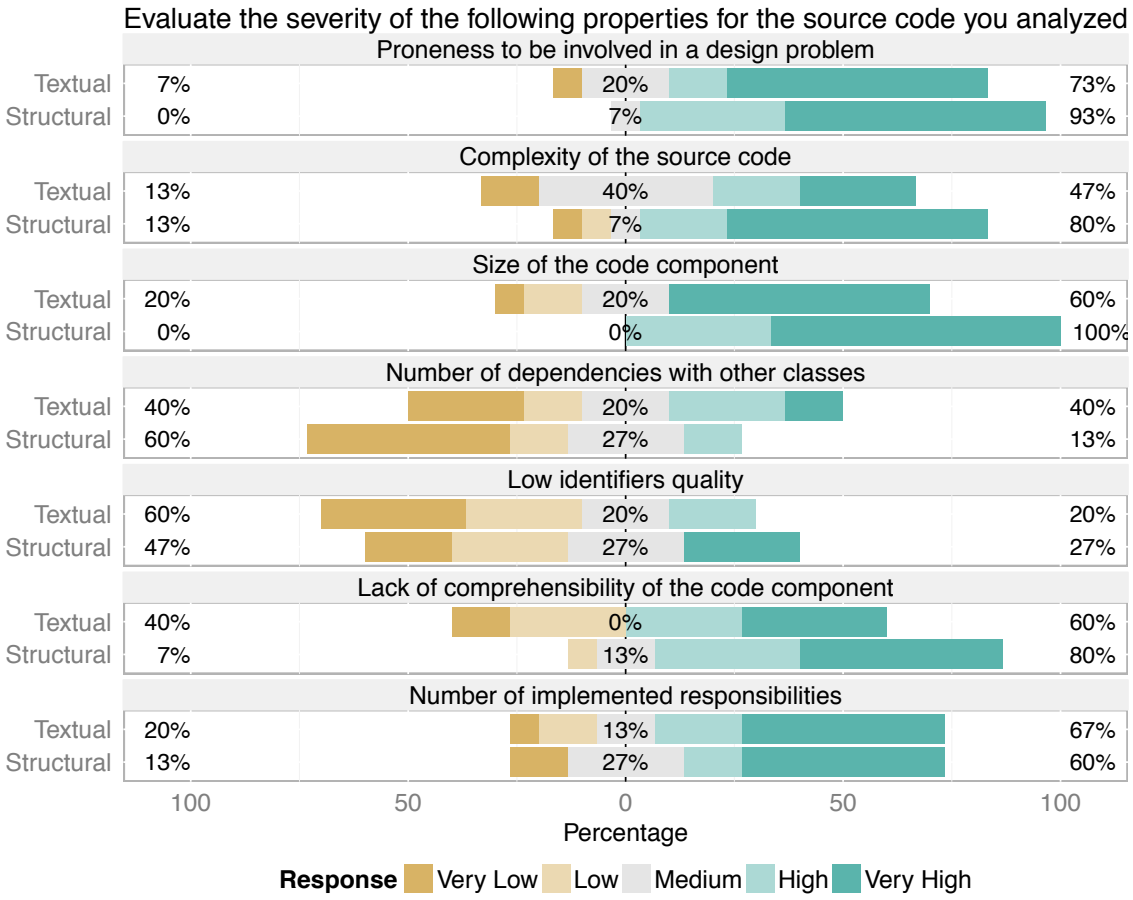


Figure 10.2: Likert chart for post-task questionnaires, where the same questions are asked at the end of each task

For most of the cases, we see that the structural instances have been marked as more severe than textual ones. While developers experience structural smells as more severe, they are not able to correctly diagnose the problems behind this type of smells because of their increased complexity. Therefore, we can affirm that textual properties are useful in the detection phase in order to ease the comprehension and analysis of a design flaw. At the same time, considering the refactorings that participants suggested, we see there is the need for tools providing better support in the refactoring of structural code smells.

When considering properties as code size and number of responsibilities, we did not find any statistical difference between structural and textual smells. How-

ever, in the open discussion session, the quality experts observed that "*size and number of responsibilities are quite related to each other*" when the goal is to understand whether a specific code unit is affected by a smell or not. Indeed, they observed that if a class is too large this is a first symptom for the class to be a potential *Blob*. However, they also noted that "*raw number of lines is not the end of the story because developers have to read the code*", build their own mental model and then "*figure out whether the class or the method implements too many responsibilities, which is the main reason of other issues*". Moreover, the quality experts observed that "*the number of lines in a method and the presence of unrelated parameters are the two aspects to look at*" when identifying code smells at method level, where unrelated parameters denote "*parameters that are interpreted by developers as unrelated looking at the identifier names*". Finally, for code smells at the class level they mentioned that "*the number of concepts that can be derived by method names and attributes*" is the main source of information while the raw number of lines is not a good approximation of the phenomenon. Through this analysis we learnt that the size of a code unit does not always represent a good measure to identify design flaws, since to better evaluate the presence of spurious responsibilities the analysis of textual components is needed.

Participants also evaluated the severity of two other *dimensions* for the code components analyzed, namely complexity and comprehensibility. From Figure 10.2 we can observe that participants perceived the structural code smells as more complex (80%) with respect to textual smells (47%). The Wilcoxon test reveals that such difference is marginally significant (ρ=0.057) with a negative *medium* Cliff's d effect size (d=-0.36). Similarly, for comprehensibility we observe that 80% of participants labeled the structural code smells as difficult to comprehend (high or very high severity) with respect to 60% of participants for textual smells. Such a difference is statistically significant with ρ=0.029 and has a negative *small* Cliff's d effect size (d=-0.275). This result highlights that structural smells are perceived as more difficult to deal with.

To practically understand the effect of these differences, during the open discussion we asked the quality consultants to directly compare pairs of code compo-

nents (*e.g.,* textual vs structural blobs) in order to illustrate possible steps for the application of a refactoring. All the experts stated that in each pair there was "*one instance much more complex to understand, which makes difficult the derivation of a precise refactoring operation that should be applied*". For example, *Expert1* reported that "*looking at the raw number of decision points*", the two methods `addAttributes` from the class `ClassFile` (structural long method) and `set` from the class `Option` (textual long method) "*seem to be equally complicated"*, but by looking more carefully at the code, the method `set` *"is instead well structured because there is a pattern in the if conditions. For this method it is easier to imagine that potential way to fix the problem is to write different methods for different if conditions*". While the structural long method `addAttributes` from class `ClassFile` "*is very complicated and deciding the refactoring operations to apply is not so simple*". As another example, *Expert4* reported that the method `explain` from class `DisjunctionMatchQuery` "*is quite complicated despite its length: it contains too many concepts and some parameters are simply passed to other methods*". It is worth noting that this method contains only 17 lines of code and it is detected as *Feature Envy* by TACO (textual tool). The structural *Feature Envy*, *i.e.,* method `nextChar` from `HTMLStripCharFilter`, is perceived "*more complex to manage because it contains too many responsibilities spread across external classes; it implements a state machine with no meaningful names to help the comprehension*". Thus, we can affirm that the higher comprehensibility of textual code smells seems to help developers in finding appropriate refactoring solutions. The result is in line with what we found studying the evolution of textual smells, *i.e.,* they are generally more prone to be refactored, as well as more prone to be subject to activities aimed at reducing their intensity over time.

10.6.3 Threats to Validity

The main threat related to the relationship between theory and observation (*construct* validity) is represented by the subjectivity of textual and structural code smell perception. To limit this bias, we carefully selected the study participants by recruiting developers having industrial experience as well as a quite long career in

software development. Moreover, we complement our analysis by involving five quality consultants having a solid knowledge about software quality and code smells.

As for the factors potentially influencing our findings (threats to *internal* validity), we ensured that participants were not aware of the types of code smells affecting the provided instances nor the underlying techniques used for detection.

Finally, as for the generalizability of our results (threats to *external* validity), possible threats can be related to the set of chosen objects and to the pool of the participants in the study. Concerning the chosen objects, we are aware that our study is based on smell instances detected in two Java systems only, and that further studies are needed to confirm our results. In this study we had to constrain our analysis to a limited set of smell instances, because the task to be performed by each respondent had to be reasonably small (to ensure a decent response rate). As for the participants, we involved 10 industrial developers. While a replication of the study aimed at corroborating the results achieved could be worthwhile, the developers involved have a strong experience in the development. At the same time, we complement our analyses by involving 5 quality consultants having a solid knowledge about software quality and code smells.

10.7 Conclusion

In this chapter, we conducted a systematic investigation aimed at analyzing how developers perceive and act on code smell instances depending on which source of information is used for the detection, *i.e.*, structural metrics versus textual data. To this aim, we mined historical data from 301 releases and 183,514 commits of the 20 open source projects in order to monitor developers' activities on textual and structural code smells over time. We also conducted a qualitative study with industrial developers and software quality consultants to analyze how they perceive and react to code smell instances of different nature. The results of the study highlight four key findings:

1. Textually detected code smells are generally perceived by industrial developers as actual design problem equally dangerous to structural smells, even if they do not exceed any structural metrics' thresholds.

2. Textual and structural smells are treated differently: the intensity of the former tends to decrease over time, while the intensity of latter tends to increase.

3. Textually detected smells are more prone to be resolved through refactoring operations or enhancement activities.

4. Textual smells are perceived as easier to understand. As a consequence, accurate refactoring operations can be derived more quickly.

Our findings confirm that software metrics are not the unique source of information that developers use to evaluate the quality of source code [46, 212, 213]. Moreover, the results achieved represent a call to arms for researchers and practitioners to investigate the human's perspective for building a new generation of code smell detectors and refactoring tools. Our future agenda includes, indeed, the definition of a new set of hybrid techniques that efficiently use both sources of information to detect code smells.

On the other hand, our results shed light on an important motivation behind the lack of refactoring activities performed to remove code smells [14, 49, 40], *i.e.*, it seems developers are not able to practically work on structurally detected code smells. Therefore, our results clearly highlight the need of having techniques and tools able to help developers in refactoring code smells characterized by structural design flaws, which are difficult to understand for developers.

Part 4

Conclusion and Further Research Directions

Chapter 11

Lessons Learnt and Open Issues

The issues related to the management of code smells have attracted an ever increasing attention by the research community, interested in studying the dynamics behind code smell evolution as well as of their detection and removal. Although several steps ahead have been done in recent years, in the context of this thesis we highlighted three specific limitations of previous research, *i.e.,* (i) absence of empirical evidence about code smell introduction, (ii) the relevance and the impact of code smells on maintainability was studied through small-scale investigations, and (iii) existing code smell detectors were inadequate for the detection of many smells. To overcome the limitations found in the current research, we proposed a number of empirical investigations and novel approaches for detecting code smells. We firstly investigated when and why developers introduce code smells in practice (Chapter 3), by conducting a large-scale empirical study on the evolution history of 200 systems. The findings sometimes contradicted common winsdom, showing that (i) code smells are usually introduced during the first commit of an artifact rather than be the result of several maintenance activities applied on an artifact, (ii) the code artifacts becoming smelly during the evolution are characterized by peculiar metrics' trends that are different from those of the artifacts that will not become smelly, (iii) refactoring can be the cause of smell introduction, and (iv) high workload and release pressure are the main reasons for the introduction of smells.

At the same time, we studied the longevity of code smells as well as how developers generally remove them from the source code (Chapter 3). We found that code smells have a high survivability and are rarely removed as a direct consequence of refactoring activities. Moreover, we also found that refactoring is often non-efficient for smell removal (Chapter 6). Then, we conducted a large-scale empirical study on the impact of design flaws on change- and fault-proneness (Chapter 4). In general, we confirmed previous findings on the relationship between smells and change-proneness. At the same time, we found that smells seem to be not the direct cause of fault-proneness but rather a co-occurring phenomenon in some parts of the system that are intrinsically fault-prone for various reasons.

Finally, we studied the developers' perception of code smells (Chapter 5), finding that most of the design flaws characterized by complex/long code are perceived and correctly identified by developers, while for other smells depend on their intensity. In the second place, we defined two new approaches for smell detection, based on the use of alternative sources of information.

The first one, coined HIST (Chapter 7), is able to detect code smells following a two-step approach: in particular, the change history of a software system is extracted in order to collect the fine-grained changes, and then a set of heuristics based on co-change or frequency analysis are adopted to detect instances of five code smells, *i.e.*, *Divergent Change*, *Shotgun Surgery*, *Parallel Inheritance*, *Blob*, and *Feature Envy*. The evaluation of the approach has been conducted on 20 open-source systems, where we evaluated (i) the precision and the recall of the approach achieved when comparing the results of the technique with a manually-built oracle, and (ii) the complementarity between HIST and other existing code smell detectors solely based on structural analysis. The results highlighted that our historical-based approach achieves a precision between 72% and 86%, and a recall between 58% and 100%. Moreover, we noticed that often our approach outperformed the baseline approaches which was compared to. More importantly, however, is that the results provided by historical and structural approaches are complementary, *i.e.*, the two types of information can detect different smell instances, paving the way to a combined approach able to increase the overall precision and

recall of the single approaches.

Since a tool is useful in practice only if its results are meaningful for developers, we employed HIST in a second empirical study (Chapter 8), where we compared the developers' perception about code smell instances retrieved by both the historical and the structural-based approaches. Interestingly, more than 75% of the smells identified through HIST were considered as actual design flaws by developers. In addition, smell instances identified by both HIST and the structural-based techniques are the ones that perfectly match developers' perception of design flaws, indicating once again the need of combined detection approaches.

Besides HIST we also defined TACO, an approach for detecting code smells using Information Retrieval techniques. Specifically, it is able to identify a set of code smells characterized by promiscuous responsabilities, *i.e., Long Method*, *Blob*, *Promiscuous Package*, *Feature Envy*, and *Misplaced Class*. In this case, the detector firstly extracts the source code to analyze in order to normalize it through a standard IR normalization approach. Then, detection rules based on the textual similarity between the code elements contained in a class (*e.g.,* methods) or in a package (*e.g.,* classes) are applied. As done previously, we involved TACO in a preliminary evaluation where we ran the technique against a manually-produced oracle referring to 20 open source systems to get hints about its precision and recall. At the same time, we compared our textual-based technique with existing structural-detectors. The results reported a precision ranging between 67% and 72%, while a recall ranging between 70% and 85%. Furthermore, most of the times our technique outperformed other structural-based techniques, denoting an interesting complementarity in the set of smell instances detected. Such result opens the road to a further combination between different types of sources of information.

Once achieved good results in terms of accuracy, we employed TACO and the structural-based baselines in two other experiments (Chapter 10). In the first one, we set up a software repository mining study to analyze how the degree of smelliness of textually or structurally detected smells change in different ways over the history of a software system. In the second place, we conducted a user study with

industrial developers and quality experts in order to qualitatively analyze how they perceive code smells identified using the two different sources of information. Both the studies reported similar results: textually detected smells are easier to perceive, and for this reason they are considered easier to refactor, and actually refactored more often than smells detected using structural properties.

Last but not at least, we ensure the technological transfer and the replication of the results by making a number of tools, replication packages, and datasets publicly available.

11.1 Lessons Learnt

The main lessons learnt from this thesis can be summarizable as follow.

- **Lesson 1.** *Code smells are introduced as the source code of a class is committed for the first time in the repository.* One of main results of this thesis consists of the understanding of the state-of-the-practice of code smell introduction and removal. Unlike to what usually thought in the research community, we learnt that code smells are not introduced because of subsequent activities, but rather when developers are working on a class for the first time. Therefore, the code smell detectors defined so far might be somehow biased by the wrong theories on smell introduction. At the same time, the result highlights the need of having code quality checkers that better assist developers during their activities (*e.g.*, smell detectors working at commit-level or incorporated in the code review process).

- **Lesson 2.** *Code smells are a threat for the maintainability of source code.* As reported throughout the empirical studies reported in the thesis, code smells have a high diffuseness over real applications and contribute to cause maintainability issues such as making an affected class more change- and fault-prone. Thus, code smells are not a theoretical problem, and efficient research solutions need to be carried out in order to support developers in their daily activities. Similarly, this result highlights the need for software developers

to constantly monitor the quality of source code in order to avoid the introduction of technical debt causing high unforeseen costs during the evolution of systems.

- **Lesson 3.** *Not all the code smells are perceived as actual design problems, but textual and historical analysis can support their correct diagnosis.* In this thesis, we focused a lot on the developers' perspective, since we were interested in understanding the underlying properties of code smells. As a result, we discovered that only a small subset of code smells fit the developers' perception of design flaws: as easily imaginable, these are the smells characterized by long or complex code. At the same time, the correct identification of code smells that highlight violations to Object-Oriented programming mostly depend on the intensity of the problem. However, an even more important finding regards the role of textual and historical analysis from the developers' side. Indeed, we found that smells detected using HIST and TACO are better aligned with what developers consider a real problem. This is because (i) the more a code smell change over time the more a developer can recognize that smell as a recurrent problem [12], (ii) textual issues are closer to the conceptual understanding of design flaws [54]. Based on such lesson, future detectors should rely on alternative sources of information in order to output sets of code smells that are close to the developers' perception of design problems.

- **Lesson 4.** *Code smells characterized by long and/or complex code are the ones causing more issues for software maintenance and evolution*. Among all the code smells defined in literature, the ones referring to complex and/or long code (*i.e., Blob*, *Complex Class*, *Spaghetti Code*) represent the major threat to the maintainability of software systems. Even more interestingly, the refactoring operations applied on these smells allow gaining more in terms of ease of future maintenance (as shown in Chapter 4). Other smells, instead, seem to be practically poorly relevant since they are neither perceived by developers

nor they cause serious maintainability issues. Thus, the research community should devote further effort in the definition of effective strategies for the identification and the removal of long and/or complex smells. As for software developers, this result highlights the need of adopting code quality checkers able to prevent the introduction and/or the growth of design flaws.

- **Lesson 5.** *Developers rarely remove code smells and, when they do, refactoring is not the main strategy applied.* Even though the harmfulness of code smells, this thesis also shed lights on a relevant practical issue: in general, developers do not refactor code smells. On the one hand, this is due to the fact the code smells are diffucult to understand or not correctly perceived as design flaws by developers, and as such, developers cannot simply design proper refactoring solutions. On the other hand, we found that most of the times code smells are removed because of collateral activities, such as a major restructuring of a class. At the same time, when studying the properties of textual smells, we observed that developers can easily find ways to improve the design of textually detected code smells rather than the one of structural smells. Therefore, the use of textual information for smell detection is highly recommendable, since it helps developers in correctly diagnose a design flaw.

- **Lesson 6.** *Refactoring may introduce code smells.* Even though most of the smells are introduced when developers are working on the implementation of new features or when enhancing existing parts of a system, a non-negligible portion of smells are introduced because of refactoring operations applied by programmers in order to re-organize the source code. While this finding was somehow unexpected given the nature of refactoring, it highlights the need of tools able to support developers in correctly assess the impact of refactoring operations on source code before their actual application. At the same time, this result clearly demonstrates how software developers should actively use automatic refactoring tools during their daily activities.

- **Lesson 7.** *Structural-based code smell detection is not bulletproof, and alternative*

sources of information can complement existing approaches. Another important finding reported in this thesis regards the role of the analysis of the structural properties of source code for smell detection. Indeed, while we acknowledge that existing detectors obtain good performances in terms of precision and recall, it is also true that many code smells cannot be adeguately detected using code metrics. Conversely, the approaches proposed in this thesis can nicely complement structural-based techniques. Indeed, both historical and textual approaches (i) can achieve better results in term of precision and recall than existing approaches, and (ii) are better complied to the developers' mental model. These findings highlight the need of further studies and techniques able to complement different sources of information.

These lessons will drive future research in the field. Some specific further envisioned contributions are described in the subsequent section.

11.2 Open Issues

Despite the effort devoted by the research community and despite the advances proposed in this thesis, the detection of code smells still propose a number of open issues and challenges that need to be addressed in the future.

- **Open Issue 1.** *Toward combined code smell detectors.* As highlighted in our work, different techniques capture different smells using different types of information [52, 53]. Thus, to obtain a technique that significatively improves the current state-of-the-art smell detectors, a combination of different sources of information is needed. However, such a combination is not trivial. For example, during the development of TACO [53], we evaluated a simple combination between textual and structural information obtained using AND/OR operators for the detection of the *Feature Envy* smell: in the AND case we experienced a strong increase of the precision (*i.e.*, +17% than TACO, +27% than JDeodorant), accompanied by a strong decreases of the recall (-34% than TACO, -31% than JDeodorant). Similarly, in the OR case the

recall strongly increases (+24% than TACO, +34% than JDeodorant), while the precision decreases of almost 39% for TACO and 36% for JDeodorant. A possible envioned way to combine different types of information is the introduction of a mechanism for *local* smell detection. Similarly to what designed in other contexts, such as bug prediction [214], the idea would be that of clustering the classes of a software system in homogeneous groups, and then applying a different type of code smell detection based on the properties of the classes in each cluster. For instance, let suppose that two clusters have different characteristics with respect to the number of changes they experienced in the past. In this case, a historical approach may work properly on the cluster containing classes changing more while it may have poor performances on the other cluster, where a structural-based approach may achieve better performances.

- **Open Issue 2.** *Improving the quality of recommendations.* As recently pointed out [40, 49], only a small percentage of code smells is actually removed by developers. In our opinion, the reason behind this data is twofold. First of all, since the removal of code smells is a time-consuming and error-prone task [124], it is important that such techniques find relevant suggestions about which parts of the source code a developer should care about. To this aim, the research community needs to focus its attention on how to rank code smells based on their importance for developers and/or the context a developer is working on. While some attempts in this direction have already carried out [39, 38], we believe that these aspects need to be still improved in next years. Secondly, as revealed in our research on how code smells are introduced [47], code components are generally affected by bad smells since their creation. This means that new recommenders implementing a *just-in-time* philosophy would be worthwhile. On the other hand, we found that there are also several cases in which code smells are introduced as consequence of several maintenance activities performed on a code artifact. In these cases, such code components are characterized by peculiar metrics trends, differ-

ent from those of clean artifacts. This implies the possibility to define a new generation of recommenders able to *predict* which classes will become smelly over time and, therefore, allow a more suitable way to manage code smells.

- **Open Issue 3.** *Improving the usability of code smell detectors.* Detection tools might require the definition of several parameters. Thus, they might be hard to understand and to work with, making developers more reluctant to use such tools. In addition, it is necessary to define a good strategy for the visualization and the analysis of candidate smells. This issue is particular important since the smells identified by any detection tool need to be validated by the user. Thus, a good graphic metaphor is required to highlight problems to the developers eye, allowing her to decide which of the code components suggested by the tool really represent design problems. The problem of usability is particularly important in new development contexts, such as the mobile apps development, where apps are developed by both senior and novice programmers.

- **Open Issue 4.** *Contextualize the research on code smells.* The empirical studies presented in this thesis, as well as most of the previous research in the field of code smells do not consider the quality practices used by programmers during their daily activities. For instance, the systematic adoption of code reviews [215] or static analysis tools [216] may significantly influence the quantity of code smells introduced by developers. At the same time, a deeper analysis of these practices might provide further useful insights about the motivations which push developers in introducing technical debt in the source code.

Among the three open issues, the main goal is that to devise combined techniques and prioritization approaches, which would be a direct follow-up of the work presented in this thesis.

11.3 Further Research Directions on Code Smells

While some future challenges are related to the way current code smell detectors work, other important challenges refer to the usefulness of smell-related information in other software engineering research areas. In the following, we delineate the future research directions and report some preliminary analyses that we already carried out to face the experienced issues.

11.3.1 Code Smells in Test Code

Testing represents a significant part of the whole software development effort [217]. When evolving a software system, developers evolve test suites as well by repairing them when needed and by updating them to sync with the new version of the system. To ease developers' burden in writing, organizing, and executing test suites, nowadays appropriate frameworks (*e.g.,* JUnit [217])—conceived for unit testing but also used beyond unit testing—are widely adopted.

While in the context of other code artifacts (in the following referred to as "production code") researchers have provided (i) definitions of symptoms of poor design choices, known as "code smells" [8], for which refactoring activities are desirable, (ii) automated tools to detect them (*e.g.,* [25, 27]), and (iii) evidence of the impact of code smells on program comprehensibility and maintainability, a little knowledge is available with regard to a quite related phenomenon occurring in test suites, *i.e., test smells*. Test smells—defined by van Deursen *et al.* [218]—are caused by poor design choices (similarly to code smells) when developing test cases: the way test cases are documented or organized into test suites, the way test cases interact with each other, with the production code and with external resources are all indicators of possible test smells. For instance, *Mystery Guest* occurs when a test case is using an external resource, such as a file or a database (thus, making the test not self-contained), and *Assertion Roulette* when a test case contains multiple assertions without properly documenting all of them [218].

Empirical studies have shown that test smells can hinder the understandabil-

ity and maintainability of test suites [110, 111], and refactoring operations aimed at removing them have been proposed [218]. At the same time, the research community is extremely active in the field of automatic test case generation, where complete test suites are generated automatically using, for instance, random strategies [219] or search-based algorithms [112, 220].

Nevertheless, it is still not clear how developers perceive test smells and if they are aware of them at all. Also, it is not known whether test smells are introduced as such when test suites are created, or if test suites become "smelly" during software evolution, and whether developers perform any refactoring operations to remove test smells. Finally, it is unclear the role of test smells in automatically generated test code.

Such information is of paramount importance for designing smell detection rules and building automated detection tools to be incorporated in the development process, and especially in the continuous integration processes [221], where automated tools could identify test smells and, because of that, make the build fail and notify developers about the presence of the test smells. Highlighting test smells in scenarios where it is known that developers do not want and need to maintain them—*e.g.*, because there is no better solution—would make automated smell detection tools usable, avoiding recommendation overload [222] and even build failures.

In this context, we started studying some aspects related to the phenomenon, and in particular we focused our attention on (i) the lifecycle of test smells, (ii) the diffusion of test smells in automatically generated test code, and (iii) the improvement of automatic test case generator through the use of quality metrics. A brief summary of the results achieved as well as future goals is provided in the following.

The Lifecycle of Test Smells

We conducted a thorough empirical investigation into the perceived importance of test smells and of their lifespan across software projects' change histories. Specif-

ically, we focused our attention on six test smell types from the catalogue by Van Deursen *et al.* [218], *i.e., Assertion Roulette*, *Eager Test*, *General Fixture*, *Mystery Guest*, and *Sensitive Equality*.

First, we conducted a survey with 19 developers assessing whether developers could recognize instances of test smells in software projects. Such a survey obtained a clear negative result, indicating that, unlike what previously found for code smells [50], there is basically no awareness about test smells, highlighting the need for (semi-) automatic support to aid in detecting these design issues.

Thus, we conducted a mining study over the change history of 152 software projects to gather the deeper knowledge needed to design effective test smells detectors. In the context of the study, we investigated (i) when test smells are introduced; (ii) how long test smells survive (and whether developers try to remove them); and (iii) whether test smells are related to the presence of smells in production code, and, therefore, there can be synergies in their detection. The achieved results indicate that (i) test smells mostly appear as the result of bad design choices made during the creation of the test classes, and not as the result of design quality degradation over time, (ii) test smells stay in the system for a long time, with a probability of 80% that a test smell would not be fixed after 1,000 days from its introduction, and (iii) complex classes (*e.g.,* Blob classes) in the production code are often tested by smelly test classes, thus, highlighting a relationship existing between code and test smells.

The results achieved so far represent the main input for our future research agenda on the topic, mainly focused on designing, developing, and evaluating a new generation of code quality-checkers, such as *just-in-time* refactoring tools able to perform quality checks at commit time or even while the code is written in the IDE, recommending to the developer how to "stay away" from bad design practices. Moreover, we will study whether software systems using specific code quality (*e.g.,* pull request) and testing strategies (*e.g.,* systems testing both at unit and integration levels) are more aware about test smells than other projects.

The Diffuseness of Test Smells in Automatically Generated Test Code

We conducted an empirical investigation on the diffuseness of test smells in the JUnit test classes automatically generated by EVOSUITE [112] on 110 open source software projects from the SF110 Corpus of Classes [223]. The study aimed at assessing (i) to what extent test smells are diffused in automatically generated test classes, (ii) which test smells tend to occur and co-occur more frequently, and (iii) if there exist a relationship between the presence of test smells and the project characteristics. Overall, we analyzed the behavior of eight test smells presented in literature [218].

Results indicate that (i) test smells are largely diffused, *i.e.,* 83% of JUnit classes are affected by at least one test smell; (ii) the *Assertion Roulette* test smell is the most frequent one (contained in 54% of classes), followed by *Test Code Duplication* and *Eager Test* (contained in 33% and 29% of JUnit classes, respectively); (iii) all the test smells frequently co-occur with *Assertion Roulette*; (iv) three pairs of smells, namely *Mystery Guest* and *Resource Optimism*, *Mystery Guest* and *Indirect Testing*, and *Indirect Testing* and *Test Code Duplication* tend to co-occur quite frequently; and (v) all the test smells but *For Testers Only* and *Indirect Testing* have strong positive correlations with structural characteristics of a system, such as size and number of classes in the project.

These lessons will drive future research in the field, which will be focused on the designing of new algorithms that, on the one hand, try to balance branch coverage criteria with smell-related information and, on the other hand, try to automatically create text fixtures for test cases generated using existing tools.

Putting Code Quality in the Process of Automatic Test Case Generation

Conventional approaches to test case generation mainly focus on code coverage as a unique goal to achieve, without taking into account other factors that can be relevant for testers. For example, Afshan *et al.* [224] highlighted that one such factor is the effort needed to manually check test data input and test results (*e.g.,* assertions) in order to assess whether the software behaves as intended. Therefore, they

have incorporated language models into the data generation process with the aim of generating natural language like input strings to improve human readability.

Recently, Daka *et al.* [225] used a post-processing technique to optimize readability by mutating generated tests leveraging a domain-specific model of unit test readability based on human judgement. Other non coverage-based criteria exploited in literature for test case generation include execution time [226, 227], memory consumption [228], test size [229, 230, 220], and ability to reveal faults [227].

Nevertheless, none of them explicitly consider *test code quality* metrics as an objective to reach besides code coverage. Poorly designed tests are known to have a negative impact on test maintenance, as they are more difficult to adjust when production code changes [231, 232, 233]. Automated tests first need to be maintained when they are generated, since testers need to manually validate each test case to check the assertions (oracle cost) [224, 234]. In addition, tests also need to be maintained and eventually updated according to the changes performed in the production code during later development activities. Therefore, we argued that achieving easily maintainable tests is a desirable and important goal in test case generation. The related literature provides a plethora of metrics to detect poorly designed tests, such as rules for test smells detection [235, 218, 236]. In the context of our work, we consider two simple, yet critical quality metrics for evaluating test code maintainability, namely *test cohesion* and *test coupling*.

For measuring test cohesion and test coupling we rely on Information Retrieval (IR) methods, similarly to previous papers for assessing the quality of production code [45, 237, 238]. Specifically, we define two novel metrics, namely *Coupling Between Test Methods* (CBTM) and *Lack of Cohesion of a Test Method* (LCTM) inspired by *conceptual coupling* and *conceptual cohesion*, which are two well-known metrics to assess code quality [238]. We choose IR methods since previous studies [45, 237, 238] demonstrated how textual analysis often outperforms structural metrics in its ability to describe cohesion and coupling phenomena.

To evaluate to what extent automatically generated test cases present design problems, we conducted a large scale preliminary study on the SF110 dataset [223] and using EVOSUITE [229] as test case generation tool, which was aimed at evalu-

ating to what extent automatically generated test cases present design problems. This analysis revealed that most automatically generated tests suffer from high coupling with other tests in the same test suite. Moreover, up to 28% of test cases (test methods in JUnit) suffer from low cohesion.

Given the results of this exploratory analysis, we proposed to incorporate our quality metrics LCTM and CBTM into the main loop of EVOSUITE to guide the search toward more cohesive and less coupled tests. For this, we extended the MOSA algorithm, a many-objective genetic algorithm recently proposed [220], by incorporating our quality metrics within the selection mechanism. To evaluate our quality-based variant of MOSA, we conducted a second empirical study on 43 randomly sampled classes from the SF110 dataset, aimed at analyzed three different aspects. First, if the modified MOSA algorithm is able to actually produce more cohesive and less coupled tests. In the second place, we assessed whether the quality optimization affect the branch coverage achieved by the test cases. Finally, we verified the size of the test cases generated, in order to understand if the quality optimization has also an effect on this phenomenon.

From the results, we firstly observed that the generated test cases are statistically more cohesive and less coupled. Moreover, the quality-based automatic generation process actually has a positive impact on branch coverage and test suite size. Finally, the size of the generated test cases tends to decrease, suggesting that our process can nicely complement existing post-search minimization strategies.

Our future work includes the evaluation of the impact of our quality-based algorithm on the effectiveness of test cases, as well as the evaluation of the effects on other maintainability factors, (*e.g.,* readability). Moreover, we plan to assess possible gains (if any) from the application of other test code quality metrics, as well as smell-related information in the automatic test case generation process.

11.3.2 The Role of Smell-related Information in Other SE Tasks

The second future direction is related to the evaluation of the usefulness of smell-related information in the context of other software engineering tasks. This chal-

lenge ranges from the already mentioned automatic test case generation to approaches for traceability link recovery, where some code smells such as *Blob*, may hinder the ability of automatic techniques in correctly detecting links between artifacts.

To preliminary investigate our hyphotheses, we focused our attention on the role of code smells in bug prediction. Indeed, the impact of code smells on the bug-proneness of classes has been demostrated in several studies [17, 41, 44], including the one presented in Chapter 4. Although these studies showed the potential importance of code smells in the context of bug prediction, such observations have been only partially explored by the research community. A prior work by Taba *et al.* [239] defined the first bug prediction model that includes code smell information. In particular, they defined three metrics, coined as *antipattern metrics*, based on the history of code smells in files and able to quantify the average number of antipatterns, the complexity of changes involving antipatterns and their recurrence length. Then, a bug prediction model exploiting antipattern measures besides structural metrics was devised and evaluated, showing that the performances of bug prediction models can increase up to 12.5% when considering design flaws.

In our analysis, we conjectured that *taking into account the severity of a design problem affecting a source code element in a bug prediction model can contribute to the correct classification of the bugginess of such a component*. We believe that a measure of severity of code smells can provide useful insights about the proneness of classes in being buggy. To verify this conjecture, we exploited the intensity index defined by Arcelli Fontana *et al.* [106] to build a bug prediction model that takes into account the presence and the severity of design problems affecting a code component. Specifically, we evaluated the predictive power of the intensity index by adding it in (i) bug prediction model based on structural quality metrics [240], and (ii) three prediction models based on process metrics, *i.e.*, the Basic Code Change Model devised by Hassan [7], the Developer-based Model proposed by Ostrand *et al.* [241], and the Developer Changes Based Model defined by Di Nucci *et al.* [242]. On a set of 45 systems, we compared the accuracy of the models

that include the intensity index with the models that do not use any smell-related information. We also quantified the gain provided by the addition of the intensity index with respect to the other metrics used in the experimented models. Finally, we perform an empirical comparison of the performances achieved by our model and by the model suggested by Taba *et al.* [239].

The results indicated that the addition of the intensity index as predictor of buggy components generally increases the performance of structural-based baseline bug prediction models, but also highlight the importance of considering the severity of code smells in process metrics-based prediction models, where we observed improvements up to 47% in terms of F-Measure. Moreover, the models exploiting the intensity index obtain performances up to 16% higher than models built with the addition of antipattern metrics [239]. However, we observed interesting complementarities between the set of *buggy and smelly* classes correctly classified by the two models that pushed us in investigating the possibility of a combined model including product, process, and smell-related metrics. As a result, we built a *smell-aware* combined model that use a mix of product, process, and smell-related information. It provides a consistent boost in terms of prediction accuracy (*i.e.*, F-Measure) of +13% with respect to the best performing model.

Since our study has focused on *global* bug prediction, future effort will be devoted to the analysis of the contribution of smell-related information in the context of *local-learning* bug prediction models [214], as well as in the context of *change prediction*. Finally, our future research agenda includes the definition of new factors influencing the performances of bug prediction models.

11.3.3 The Impact of Code Smells on Other Non Functional Attributes

Several studies in the literature analyzed the impact of code smells on maintainability [17, 19], however the presence of code smells can negatively impact other non-functional attributes, such as performances or reliability. Analyzing these relationships might help in devising more usable tools (*e.g.*, a prioritization system

may rank code smells based on a compromise between quality and performance improvement), but also in making more prone developers to refactor code smells.

We start facing the problem by studying the impact of code smells on the energy consumption of mobile applications. In particular, we focus our attention on energy efficiency since it is becoming a major issue in modern software engineering, as applications performing their activities need to preserve battery life. Although the problem is mainly concerned with hardware efficiency, in the recent past researchers successfully demonstrated how even software may be the root of energy leaks [243]. For instance, Sahin *et al.* [244] highlighted the existence of design patterns that negatively impact the power efficiency, as well as the role of code obfuscation on the phenomenon [245].

Although several important research steps have been made and despite the ever increasing number of empirical studies aimed at understanding the reasons behind the presence of energy leaks in the source code, we observed that the research community has not considered yet the potential impact on energy consumption of a specific set of code smells defined by Reimann *et al.* [246] for Android mobile applications.

We filled this gap by studying (i) to what extent code smells, affecting source code methods of mobile applications, influence energy efficiency and (ii) whether refactoring operations applied to remove them are able to reverse the negative effects of code smells on energy consumption. In particular, our investigation focuses on 9 method-level code smells specifically defined for mobile applications by Reimann *et al.* [246] in the context of 60 Android apps belonging to the dataset provided by Choudhary *et al.* [247]. While Reimann *et al.* theoretically supposed the existence of a relationship between these code smells and non-functional attributes of source code (*e.g.*, energy consumption), to the best of our knowledge we conducted the first study aimed at practically investigating the actual impact of such code smells on energy consumption and quantifying the extent to which refactoring code smells is beneficial for improving energy efficiency.

The results of this study provided two main results. In the first place, methods affected by smells consume up to 385% more with respect to smell-free methods.

Even if the interaction between them results in lower efficiency, we found four particular smell types that frequently co-occur and that impact energy consumption more than others, *i.e., Leaking Thread*, *Member Ignoring Method*, *Slow Loop*, and *Internal Getter and Setter*. These aspects highlight the importance of investing (i) in studying more in depth the dynamics behind *Android-specific* code smells and (ii) in developing tools that prevent their introduction.

In the second place, refactoring code smells is a key activity to improve energy efficiency. We found that the energy consumption of refactored methods is reduced by up to 900% with respect to smelly methods. Therefore, we empirically demonstrated how small changes applied to remove code smells result in applications that are much more efficient in terms of energy consumption. Approaches and tools able to support Mobile developers in automatically refactoring the source code represent a *must* for future research in the field.

These are the main points of interest for our future agenda. Indeed, we plan to futher investigate the possibility to design new code quality-checkers and refactoring tools aimed at analyzing, removing, and preventing the introduction of Android-specific code smells. Moreover, we aim at corroborating our results by studying the impact of code smells on other non-functional attributes, such as performances.

List of Publications

The complete list of publications is reported below. The ⋆ symbol highlights the publications discussed in this dissertation.

International Journal Papers

J1 - **F. Palomba**, A. De Lucia, G. Bavota, R.Oliveto. *Anti-Pattern Detection: Methods, Challenges, and Open Issues*. Advances in Computers, volume 95, pp. 201-238. ⋆

J2 - G. Bavota, A. De Lucia, M. Di Penta, R.Oliveto, **F. Palomba**. *An Experimental Investigation on the Innate Relationship between Quality and Refactoring*. Journal of Systems and Software (JSS), Volume 107, September 2015, pp. 1-14. ⋆

J3 - **F. Palomba**, G. Bavota, M. Di Penta, R.Oliveto, D. Poshyvanyk, A. De Lucia. *Mining Version Histories for Detecting Code Smells.* Transactions on Software Engineering (TSE), Volume 41, Issue 5, pp. 462-489. ⋆

J4 - M. Tufano, **F. Palomba**, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, D. Poshyvanyk. *When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away).* Transactions on Software Engineering (TSE), to appear. ⋆

J5 - **F. Palomba**, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia. *On the Distribution and the Impact on Maintainability of Code Smells: A Large Scale Empirical Investigation*. In Major Revision at the Journal of Empirical Software Engineering (EMSE). ⋆

J6 - **F. Palomba**, A. Panichella, R. Oliveto, A. Zaidman, A. De Lucia. *The Scent of a Smell: An Extensive Comparison between Textual and Structural Smells*. In Major Revision at the Transactions on Software Engineering (TSE). ⋆

J7 - **F. Palomba**, M. Zanoni, F. Arcelli Fontana, A. De Lucia, R. Oliveto. *Toward a Smell-aware Bug Prediction Model*. Submitted to the Transactions on Software Engineering (TSE). ⋆

J8 - M. Tufano, **F. Palomba**, G. Bavota, M. Di Penta, R.Oliveto, A. De Lucia, D. Poshyvanyk. *There and Back Again: Can you Compile that Snapshot?* Journal of Software: Evolution and Process (JSEP), to appear.

J9 - **F. Palomba**, M. L. Vasquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, A. De Lucia. *Crowdsourcing User Reviews to Support the Evolution of Mobile Apps*. In Minor Revision at the Journal of Systems and Software (JSS).

J10 - D. Di Nucci, **F. Palomba**, G. De Rosa, G. Bavota, R. Oliveto, A. De Lucia. *A Developer Centered Bug Prediction Model*. Transactions on Software Engineering (TSE), to appear.

J11 - D. Di Nucci, **F. Palomba**, R. Oliveto, A. De Lucia. *Dynamic Selection of Classifiers in Bug Prediction: An Adaptive Method*. Submitted to the Transactions on Emerging Topics on Computational Intelligence (TECTI).

J12 - **F. Palomba**, D. Di Nucci, A. Panichella, A. Zaidman, A. De Lucia. *On the Impact of Code Smells on the Energy Consumption of Mobile Applications*. Submitted to the Journal of Empirical Software Engineering (EMSE). ⋆

International Conference Papers

C1 - **F. Palomba**, G. Bavota, M. Di Penta, R.Oliveto, A. De Lucia, D. Poshyvanyk. *Detecting Bad Smells in Source Code Using Change History Information*. In Proceedings of the 28th IEEE/ACM International Conference on Automated

Software Engineering (ASE 2013), Palo Alto, California, 2013, 11 pages, 268-278. ⋆

C2 - **F. Palomba**, G. Bavota, M. Di Penta, R.Oliveto, A. De Lucia. *Do They Really Smell Bad? A Study on Developers Perception of Bad Code Smells*. In Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME 2014), Victoria, Canada, 2014, 10 pages, 101-110. ⋆

C3 - M. Tufano, **F. Palomba**, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, D. Poshyvanyk. *When and Why Your Code Starts to Smell Bad.* In Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE 2015), Florence, Italy, 2015, 11 pages, 403-414. ⋆

C4 - **F. Palomba**. *Textual Analysis for Code Smell Detection.* In Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE 2015) - Student Research Competition (SRC) Track, Florence, Italy, 2015, 2 pages, 769-771. ⋆

C5 - **F. Palomba**, A. Panichella, A. De Lucia, R. Oliveto, A. Zaidman. *A Textual-based Technique for Smell Detection*. In Proceedings of the 24th International Conference on Program Comprehension (ICPC 2016), Austin, USA, 2016, 10 pages, 1-10. ⋆

C6 - **F. Palomba**, M. Zanoni, F. Arcelli Fontana, A. De Lucia, R. Oliveto. *Smells like Teen Spirit: Improving Bug Prediction Performance Using the Intensity of Code Smells*. In Proceedings of the International Conference on Software Maintenance and Evolution (ICSME 2016), Raleight, USA, 2016, 12 pages, pp. 244-255. ⋆

C7 - M. Tufano, **F. Palomba**, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, D. Poshyvanyk. *An Empirical Investigation into the Nature of Test Smells*. In Proceedings of the International Conference on Automated Software Engineering (ASE 2016), Singapore, Singapore, 2016, 12 pages, 4-15. ⋆

C8 - **F. Palomba**. *Alternative Sources of Information for Code Smell Detection: Postcards from Far Away*. In Proceedings of the International Conference on Software Maintenance and Evolution (ICSME 2016) - Doct. Symp., Raleight, USA, 2016, 5 pages, pp. 636-640. ⋆

C9 - G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, **F. Palomba**. *Supporting Extract Class Refactoring in Eclipse: The ARIES Project*. In Proceedings of the 34th International Conference on Software Engineering (ICSE 2012), Zurich, Switzerland, 2012. IEEE Press. Formal Tool Demo. pp. 1419-1422.

C10 - D. Di Nucci, **F. Palomba**, S. Siravo, G. Bavota, R. Oliveto, A. De Lucia. *On the Role of Developers Scattered Changes in Bug Prediction*. In Proceedings of the 31th International Conference on Software Maintenance and Evolution (ICSME 2015), Research Track, Bremen, Germany, 10 pages, pp. 241-250.

C11 - **F. Palomba**, M. L. Vasquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, A. De Lucia. *User Reviews Matter! Tracking Crowdsourced Reviews to Support Evolution of Successful Apps*. In Proceedings of the 31th International Conference on Software Maintenance and Evolution (ICSME 2015), Research Track, Bremen, Germany, 10 pages, pp. 291-300.

C12 - **F. Palomba**, D. Di Nucci, M. Tufano, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, A. De Lucia. *Landfill: an Open Dataset of Code Smells with Public Evaluation*. In Proceedings of the 12th Working Conference on Mining Software Repositories (MSR 2015), Data Showcase Track, Firenze, Italy, 2015, 4 pages, pp. 482-485.

C13 - **F. Palomba**, M. Tufano, G. Bavota, R. Oliveto, A. Marcus, D. Poshyvanyk, A. De Lucia. *Extract Package Refactoring in ARIES*. In Proceedings of the 37th International Conference on Software Engineering (ICSE 2015), Tool Demonstration Track, Firenze, Italy, 2015, 4 pages, pp. 669-672.

C14 - **F. Palomba**, A. Panichella, A. Zaidman, R. Oliveto, A. De Lucia. *Automatic Test Case Generation: What if Test Code Quality Matters?* In Proceedings of the

International Symposium on Software Testing and Analysis (ISSTA 2016), Saarbrucken, Germany, 2016, 12 pages, 130-141. ⋆

C15 - **F. Palomba**, D. Di Nucci, A. Panichella, R. Oliveto, A. De Lucia. *On the Diffusion of Test Smells in Automatically Generated Test Code: An Empirical Study*. In Proceedings of the 9th International Workshop on Search-based Software Testing (SBST 2016), Austin, USA, 2016, 10 pages, 5-14. ⋆

C16 - **F. Palomba**, P. Salza, A. Ciurumelea, S. Panichella, H. Gall, F. Ferrucci, A. De Lucia. *Recommending and Localizing Code Changes for Mobile Apps based on User Reviews*. In Proceedings of the 39th International Conference on Software Engineering (ICSE 2017), Buenos Aires, Argentina, 2017, 12 pages, to appear.

C17 - D. Di Nucci, **F. Palomba**, A. Prota, A. Panichella, A. Zaidman, A. De Lucia. *Software-Based Energy Profiling of Android Apps: Simple, Efficient and Reliable?*. In Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER 2017), Klagenfurt, Austria, 2017, 12 pages, pp. 103-114.

C18 - D. Di Nucci, **F. Palomba**, A. Prota, A. Panichella, A. Zaidman, A. De Lucia. *PETrA: A Software-based Tool for Estimating the Energy Profile of Android Applications*. In Proceedings of the Formal Tool Demo Track of the International Conference on Software Engineering (ICSE 2017), Buenos Aires, Argentina, 2017, 4 pages, to appear.

C19 - **F. Palomba**, D. Di Nucci, A. Panichella, A. Zaidman, A. De Lucia. *Lightweight Detection of Android-specific Code Smells: The aDoctor Project*. In Proceedings of the Formal Tool Demo Track of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER 2017), Klagenfurt, Austria, 2017, 5 pages, pp. 487-491.

C20 - **F. Palomba**, R. Oliveto, A. De Lucia. *Investigating Code Smell Co-Occurrences using Association Rule Learning: A Replicated Study*. In Proceedings of the International Workshop on Machine Learning Techniques for Software Quality

Evaluation (MaLTeSQuE 2017), Klagenfurt, Austria, 2017, 6 pages, to appear.

C21 - **F. Palomba**, A. Zaidman, R. Oliveto, A. De Lucia. *An Exploratory Study on the Relationship between Changes and Refactoring*. In Proceedings of the 25th International Conference on Program Comprehension (ICPC 2017), Buenos Aires, Argentina, 2017, 10 pages, to appear.

C22 - G. Catolino, **F. Palomba**, A. De Lucia, F. Ferrucci, A. Zaidman. *Using Process Metrics for Predicting Change-prone Classes: An Empirical Assessment*. In Proceedings of the 25th International Conference on Program Comprehension (ICPC 2017), Buenos Aires, Argentina, 2017, 10 pages, to appear.

C23 - **F. Palomba**, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia. *A Large-Scale Empirical Study on the Lifecycle of Code Smell Co-occurrences*. Submitted to the 33rd International Conference on Software Maintenance and Evolution (ICSME 2017).

C24 - **F. Palomba**, A. Zaidman. *Refactoring Test Code for Refactoring Its Flakiness*. Submitted to the 33rd International Conference on Software Maintenance and Evolution (ICSME 2017).

C25 - L. Pascarella, **F. Palomba**, A. Bacchelli. *Fine-Grained Just in Time Defect Prediction*. Submitted to the 33rd International Conference on Software Maintenance and Evolution (ICSME 2017).

The permission of the copyright holders of the original publications to reprint them in this thesis is hereby acknowledged.

Bibliography

[1] G. G. Roy and V. E. Veraart, "Software engineering education: from an engineering perspective," in *Software Engineering: Education and Practice, 1996. Proceedings. International Conference*, Jan 1996, pp. 256–262.

[2] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, "Software complexity and maintenance costs," *Commun. ACM*, vol. 36, no. 11, pp. 81–94, Nov. 1993.

[3] M. M. Lehman and L. A. Belady, *Software Evolution - Processes of Software Change*. Academic Press London, 1985.

[4] W. Cunningham, "The WyCash portfolio management system," *OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.

[5] D. L. Parnas, "Software aging," in *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 16-21, 1994.*, 1994, pp. 279–287.

[6] W. Harrison, "An entropy-based measure of software complexity," *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 1025–1029, Nov. 1992. [Online]. Available: http://dx.doi.org/10.1109/32.177371

[7] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, May 2009, pp. 78–88.

[8] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[9] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 682–691.

[10] ——, "Do code smells reflect important maintainability aspects?" in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 306–315.

[11] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni, "Antipattern and code smell false positives: Preliminary conceptualization and classification," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 609–613.

[12] D. Ratiu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2004, pp. 223–232.

[13] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *Int'l Conf. Quality of Information and Communications Technology (QUATIC)*. IEEE, 2010, pp. 106–115.

[14] R. Arcoverde, A. Garcia, and E. Figueiredo, "Understanding the longevity of code smells: preliminary results of an explanatory survey," in *Proceedings of the International Workshop on Refactoring Tools*. ACM, 2011, pp. 33–36.

[15] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Assessing the impact of bad smells using historical information," in *Proceedings of the International workshop on Principles of Software Evolution (IWPSE)*. ACM, 2007, pp. 31–34.

[16] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2009, pp. 75–84.

[17] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[18] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension," in *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*. IEEE Computer Society, 2011, pp. 181–190.

[19] D. Sjoberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *Software Engineering, IEEE Transactions on*, vol. 39, no. 8, pp. 1144–1156, Aug 2013.

[20] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[21] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2004, pp. 350–359.

[22] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[23] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Proceedings of the International Conference on Quality Software (QSIC)*. Hong Kong, China: IEEE, 2009, pp. 305–314.

[24] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical signatures of antipatterns: An approach based on B-splines," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2010, pp. 248–251.

[25] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[26] M. J. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in *Proceedings of the International Software Metrics Symposium (METRICS)*. IEEE, September 2005, p. 15.

[27] G. Bavota, A. Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: An improved method and its evaluation," *Empirical Softw. Engg.*, vol. 19, no. 6, pp. 1617–1664, Dec. 2014.

[28] G. Bavota, R. Oliveto, A. D. Lucia, G. Antoniol, and Y. G. Guhneuc, "Playing with refactoring: Identifying extract class opportunities through game theory," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, Sept 2010, pp. 1–5.

[29] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Using structural and semantic measures to improve software modularization," *Empirical Software Engineering*, vol. 18, no. 5, pp. 901–932, 2013.

[30] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 671–694, July 2014.

[31] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. d. Lucia, "Improving software modularization via automated analysis of latent topics and dependencies," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, pp. 4:1–4:33, Feb. 2014.

[32] M. Kessentini, S. Vaucher, and H. Sahraoui, "Deviance from perfection is a better criterion than closeness to evil when identifying risky code," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. ACM, 2010, pp. 113–122.

[33] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A cooperative parallel search-based software engineering approach for code-smells

detection," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, Sept 2014.

[34] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, and S. Ben Chikha, "Competitive coevolutionary code-smells detection," in *Search Based Software Engineering*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 8084, pp. 50–65.

[35] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, "Code-smell detection as a bilevel problem," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, pp. 6:1–6:44, Oct. 2014.

[36] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mantyla, "Code smell detection: Towards a machine learning-based approach," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, Sept 2013, pp. 396–399.

[37] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10664-015-9378-4

[38] R. Morales, Z. Soh, F. Khomh, G. Antoniol, and F. Chicano, "On the use of developers' context for automatic refactoring of software anti-patterns," *Journal of Systems and Software (JSS)*, 2016.

[39] N. Sae-Lim, S. Hayashi, and M. Saeki, "Context-based code smells prioritization for prefactoring," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[40] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 24th International Symposium on the Foundations of Software Engineering*, 2016.

[41] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *Quality Software (QSIC), 2010 10th International Conference on*, July 2010, pp. 23–31.

[42] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of Systems and Software*, pp. 1120–1128, 2007.

[43] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjøberg, "Are all code smells harmful? a study of God Classes and Brain Classes in the evolution of three open source systems," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2010, pp. 1–10.

[44] M. Gatrell and S. Counsell, "The effect of refactoring on change and fault-proneness in commercial c# software," *Science of Computer Programming*, vol. 102, no. 0, pp. 44 – 56, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642314005711

[45] A. Marcus and D. Poshyvanyk, "The conceptual cohesion of classes," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2005, pp. 133–142.

[46] R. Buse and W. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, July 2010.

[47] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proceedings of the International Conference on Software Engineering (ICSE) - Volume 1*. IEEE, 2015, pp. 403–414.

[48] F. Palomba, G. Bavota, R. Oliveto, F. Fasano, M. Di Penta, and j. . E. y. . . Andrea De Lucia, title = On the Distribution and the Impact on Maintainability of Code Smells: A Large Scale Empirical Investigation.

[49] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *J. Syst. Softw.*, vol. 107, no. C, pp. 1–14, Sep. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2015.05.024

[50] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 101–110.

[51] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 268–278.

[52] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, May 2015.

[53] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "A textual-based technique for smell detection," in *Proceedings of the 24th International Conference on Program Comprehension (ICPC 2016)*. Austin, USA: IEEE, 2016, p. to appear.

[54] ——, "The scent of a smell: An extensive comparison between textual and structural smells," *Transactions on Software Engineering*, 2017.

[55] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[56] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, and F. Palomba, "Supporting extract class refactoring in eclipse: The aries project," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 1419–1422. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337434

[57] F. Palomba, M. Tufano, G. Bavota, R. Oliveto, A. Marcus, D. Poshyvanyk, and A. De Lucia, "Extract package refactoring in aries," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15.

Piscataway, NJ, USA: IEEE Press, 2015, pp. 669–672. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819133

[58] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Lightweight detection of android-specific code smells: the adoctor project."

[59] D. Di Nucci, A. Prota, F. Palomba, A. Panichella, A. Zaidman, and A. De Lucia, "Petra: Software-based measurement of the energy consumption of android apps."

[60] F. Palomba, D. D. Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Landfill: An open dataset of code smells with public evaluation," in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. IEEE, 2015, pp. 482–485.

[61] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1^{st} ed. John Wiley and Sons, March 1998.

[62] B. F. Webster, *Pitfalls of Object Oriented Development*, 1^{st} ed. M & T Books, February 1995.

[63] A. J. Riel, *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[64] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, no. 6, pp. 476–493, Jun 1994.

[65] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object-oriented designs: using reading techniques to increase software quality," in *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, 1999, pp. 47–56.

[66] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of extract class refactorings in object-oriented systems," *J. Syst. Softw.*, vol. 85, no. 10, pp. 2241–2260, 2012.

[67] ——, "Jdeodorant: identification and application of extract class refactorings," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. ACM, 2011, pp. 1037–1039.

[68] G. Gui and P. D. Scott, "Coupling and cohesion measures for evaluation of component reusability," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. New York, NY, USA: ACM, 2006, pp. 18–21. [Online]. Available: http://doi.acm.org/10.1145/1137983.1137989

[69] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying extract class refactoring opportunities using structural and semantic cohesion measures," *J. Syst. Softw.*, vol. 84, no. 3, pp. 397–414, Mar. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2010.11.918

[70] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 287–300, March 2008.

[71] J. Nash, "Equilibrium points in n-person games," *National Academy of Sciences of the United States of America*, vol. 36, no. 1, pp. 48–49, 1950.

[72] F. Simon, F. Steinbr, and C. Lewerentz, "Metrics based refactoring," in *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2001, pp. 30–38.

[73] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *J. Syst. Softw.*, vol. 84, no. 10, pp. 1757–1782, Oct. 2011.

[74] K. Maruyama, "Automated method-extraction refactoring by using block-based slicing," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 3, pp. 31–40, May 2001. [Online]. Available: http://doi.acm.org/10.1145/379377.375233

[75] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2008, pp. 329–331.

[76] D. Atkinson and T. King, "Lightweight detection of program refactorings," in *Software Engineering Conference, 2005. APSEC '05. 12th Asia-Pacific*, Dec 2005, pp. 8 pp.–.

[77] D. M. Blei, "Probabilistic topic models," *Commun. ACM*, vol. 55, no. 4, pp. 77–84, Apr. 2012. [Online]. Available: http://doi.acm.org/10.1145/2133806.2133826

[78] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis, "Identification of refused bequest code smells," in *Proceedings of the 29th IEEE International Conference on Software Maintenance*, 2013.

[79] A. Rao and K. Raddy, "Detecting bad smells in object oriented design using design change propagation probability matrix," in *Proceedings of the International MultiConference of Engineers and Computer Scientists*, 2008, pp. 1001–1007.

[80] A. Rao and D. Ram, "Software design versioning using propagation probability matrix," in *in Proceedings of Third International Conference on Computer Applications, Yangon, Myanmar, 2005*, 2005.

[81] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: Patterns of cloning in software," *Empirical Softw. Engg.*, vol. 13, no. 6, pp. 645–692, Dec. 2008. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9076-6

[82] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, May 2009. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2009.02.007

[83] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 87–98. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970326

[84] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 368–. [Online]. Available: http://dl.acm.org/citation.cfm?id=850947.853341

[85] I. D. Baxter, C. Pidgeon, and M. Mehlich, "Dms®: Program transformations for practical scalable software evolution," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 625–634. [Online]. Available: http://dl.acm.org/citation.cfm?id=998675.999466

[86] V. Wahler, D. Seipel, J. Wolff, and G. Fischer, "Clone detection in source code by frequent itemset techniques," in *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, Sept 2004, pp. 128–135.

[87] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002. [Online]. Available: http://dx.doi.org/10.1109/TSE.2002.1019480

[88] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the International Conference on Software Engineering*, 2010.

[89] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 30th International Conference on Software Engineering*,

ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 321–330. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368132

[90] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the 8th International Symposium on Static Analysis*, ser. SAS '01. London, UK, UK: Springer-Verlag, 2001, pp. 40–56. [Online]. Available: http://dl.acm.org/citation.cfm?id=647170.718283

[91] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, ser. WCRE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 301–. [Online]. Available: http://dl.acm.org/citation.cfm?id=832308.837142

[92] C. Liu, C. Chen, J. Han, and P. S. Yu, "Gplag: Detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06. New York, NY, USA: ACM, 2006, pp. 872–881. [Online]. Available: http://doi.acm.org/10.1145/1150402.1150522

[93] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 175–186. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568286

[94] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987. [Online]. Available: http://doi.acm.org/10.1145/24039.24041

[95] J. H. Johnson, "Identifying redundancy in source code using fingerprints," in *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, ser. CASCON '93.

IBM Press, 1993, pp. 171–183. [Online]. Available: http://dl.acm.org/citation.cfm?id=962289.962305

[96] ——, "Visualizing textual redundancy in legacy source," in *Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '94. IBM Press, 1994, pp. 32–. [Online]. Available: http://dl.acm.org/citation.cfm?id=782185.782217

[97] ——, "Substring matching for clone detection and change tracking," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '94. Washington, DC, USA: IEEE Computer Society, 1994, pp. 120–126. [Online]. Available: http://dl.acm.org/citation.cfm?id=645543.655687

[98] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings of the IEEE International Conference on Software Maintenance*, ser. ICSM '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 109–. [Online]. Available: http://dl.acm.org/citation.cfm?id=519621.853389

[99] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 176–192, Mar. 2006. [Online]. Available: http://dx.doi.org/10.1109/TSE.2006.28

[100] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of the Second Working Conference on Reverse Engineering*, ser. WCRE '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 86–. [Online]. Available: http://dl.acm.org/citation.cfm?id=832303.836911

[101] ——, "A program for identifying duplicated code," *Computing Science and Statistics*, 1992.

[102] ——, "Parameterized pattern matching: Algorithms and applications," *Journal of Computer and System Sciences*, vol. 52, no. 1, pp. 28 – 42,

1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022000096900033

[103] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2002.

[104] F. Arcelli Fontana, V. Ferme, M. Zanoni, and A. Yamashita, "Automatic metric thresholds derivation for code smell detection," in *Proceedings of the 6th International Workshop on Emerging Trends in Software Metrics (WETSoM 2015)*. Florence, Italy: IEEE, May 2015, pp. 44–53, co-located with ICSE 2015.

[105] F. Arcelli Fontana, V. Ferme, and M. Zanoni, "Poster: Filtering code smells detection results," in *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, vol. 2. Florence, Italy: IEEE, May 2015, pp. 803–804.

[106] F. Arcelli Fontana, V. Ferme, M. Zanoni, and R. Roveda, "Towards a prioritization of code debt: A code smell intensity index," in *Proceedings of the Seventh International Workshop on Managing Technical Debt (MTD 2015)*. Bremen, Germany: IEEE, Oct. 2015, pp. 16–24, in conjunction with ICSME 2015.

[107] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Proceedings of the European Conference on Software Maintenance and ReEngineering (CSMR)*. IEEE, 2012, pp. 411–416.

[108] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '09, 2009, pp. 390–400.

[109] S. Vaucher, F. Khomh, N. Moha, and Y. G. Gueheneuc, "Tracking design smells: Lessons from a study of god classes," in *Proceedings of the 2009 16th Working Conference on Reverse Engineering (WCRE'09)*, 2009, pp. 145–158.

[110] G. Bavota, A. Qusef, R. Oliveto, A. D. Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *ICSM*, 2012, pp. 56–65.

[111] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia, "On the diffusion of test smells in automatically generated test code: An empirical study," in *Proceedings of the 9th International Workshop on Search-Based Software Testing*, ser. SBST '16. New York, NY, USA: ACM, 2016, pp. 5–14. [Online]. Available: http://doi.acm.org/10.1145/2897010.2897016

[112] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 416–419. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025179

[113] N. Göde, "Clone removal: Fact or fiction?" in *Proceedings of the 4th International Workshop on Software Clones*, ser. IWSC '10. New York, NY, USA: ACM, 2010, pp. 33–40. [Online]. Available: http://doi.acm.org/10.1145/1808901.1808906

[114] S. Bazrafshan and R. Koschke, "An empirical study of clone removals," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, Sept 2013, pp. 50–59.

[115] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 187–196, Sep. 2005. [Online]. Available: http://doi.acm.org/10.1145/1095430.1081737

[116] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, "An empirical study on the maintenance of source code clones," *Empirical Software Engineering*, vol. 15, no. 1, pp. 1–34, 2010. [Online]. Available: http://dx.doi.org/10.1007/s10664-009-9108-x

[117] A. F. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 306–315.

[118] D. I. K. Sjøberg, A. F. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Trans. Software Eng.*, vol. 39, no. 8, pp. 1144–1156, 2013.

[119] I. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, and M. Shepperdd, "A controlled experiment investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 72, no. 2, pp. 129 – 143, 2004.

[120] Y. Wang, "What motivate software engineers to refactor source code? evidences from professional developers," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, 2009, pp. 413 –416.

[121] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2011.

[122] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoringchallenges and benefits at microsoft," *IEEE Trans. Softw. Eng.*, vol. 40, no. 7, 2014.

[123] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A quantitative evaluation of maintainability enhancement by refactoring," in *Software Maintenance, 2002. Proceedings. International Conference on*, 2002, pp. 576 – 585.

[124] G. Bavota, B. D. Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation*. Riva del Garda, Italy: IEEE Computer Society, 2012, pp. 104–113.

[125] K. Stroggylos and D. Spinellis, "Refactoring–does it improve software quality?" in *Proceedings of the 5th International Workshop on Software Quality*, ser. WoSQ '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 10–. [Online]. Available: http://dx.doi.org/10.1109/WOSQ.2007.11

[126] E. Stroulia and R. Kapoor, "Metrics of refactoring-based development: An experience report," in *OOIS 2001*, X. Wang, R. Johnston, and S. Patel, Eds. Springer London, 2001, pp. 113–122. [Online]. Available: http://dx.doi.org/10.1007/978-1-4471-0719-4_13

[127] G. Szoke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimóthy, "Bulk fixing coding issues and its effects on software quality: Is it worth refactoring?" in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 2014, pp. 95–104.

[128] M. Alshayeb, "Empirical investigation of refactoring effect on software quality," *Information and Software Technology*, vol. 51, no. 9, pp. 1319 – 1326, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S095058490900038X

[129] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "Balancing agility and formalism in software engineering," B. Meyer, J. R. Nawrocki, and B. Walter, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team, pp. 252–266. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85279-7_20

[130] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. L. Nord, I. Ozkaya, R. S. Sangwan, C. B. Seaman, K. J. Sullivan, and N. Zazworka, "Managing technical debt in software-reliant systems," in *Proceedings of the Workshop on Future of Software Engineering Research, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2010, pp. 47–52.

[131] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Software*, vol. 29, no. 6, pp. 18–21, 2012.

[132] E. Lim, N. Taksande, and C. B. Seaman, "A balancing act: What software practitioners have to say about technical debt," *IEEE Software*, vol. 29, no. 6, pp. 22–27, 2012.

[133] F. Shull, D. Falessi, C. Seaman, M. Diep, and L. Layman, *Perspectives on the Future of Software Engineering*. Springer, 2013, ch. Technical Debt: Showing the Way for Better Transfer of Empirical Results, pp. 179–190.

[134] A. F. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 242–251.

[135] G. C. Murphy, "Houston: We are in overload," in *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France*, 2007, p. 1.

[136] I. Neamtiu, G. Xie, and J. Chen, "Towards a better understanding of software evolution: an empirical study on open-source software," *Journal of Software: Evolution and Process*, vol. 25, no. 3, pp. 193–218, 2013.

[137] G. Xie, J. Chen, and I. Neamtiu, "Towards a better understanding of software evolution: An empirical study on open source software," *2013 IEEE International Conference on Software Maintenance*, vol. 0, pp. 51–60, 2009.

[138] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. (2014) When and why your code starts to smell bad (and whether the smells go away) - replication package. [Online]. Available: http://www.cs.wm.edu/semeru/data/code-smells/

[139] J. Rupert G. Miller, *Survival Analysis, 2nd Edition*. John Wiley and Sons, 2011.

[140] J. Corbin and A. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," *Qualitative Sociology*, vol. 13, no. 1, pp. 3–21, 1990.

[141] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: The case of Apache," in *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, 2013, pp. 280–289.

[142] W. J. Conover, *Practical Nonparametric Statistics*, 3rd ed. Wiley, 1998.

[143] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[144] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *19th International Conference on Software Maintenance (ICSM 2003), 22-26 September 2003, Amsterdam, The Netherlands*, 2003, pp. 23–.

[145] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "ReLink: recovering links between bugs and changes," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. ACM, 2011, pp. 15–25.

[146] A. Bachmann, C. Bird, F. Rahman, P. T. Devanbu, and A. Bernstein, "The missing links: bugs and bug-fix commits." in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*. ACM, 2010, pp. 97–106.

[147] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. T. Devanbu, "Don't touch my code!: examining the effects of ownership on software quality," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and 13rd European Software Engineering Conference, Szeged, Hungary, September 5-9, 2011*. ACM, 2011, pp. 4–14.

[148] E. Kaplan and P. Meier, "Nonparametric estimation from incomplete observations," *Journal of the American Statistical Association*, vol. 53, no. 282, pp. 457–481, 1958.

[149] G. Scanniello, "Source code survival with the kaplan meier," in *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ser. ICSM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 524–527. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2011.6080823

[150] M. Claes, T. Mens, R. Di Cosmo, and J. Vouillon, "A historical analysis of debian package incompatibilities," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 212–223. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820545

[151] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2009.

[152] D. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures.*, second edition ed. Chapman & Hall/CRC, 2000.

[153] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research, October 27-30, 2008, Richmond Hill, Ontario, Canada*. IBM, 2008, p. 23.

[154] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. IEEE / ACM, 2013, pp. 392–401.

[155] O. Chaparro, G. Bavota, A. Marcus, and M. Di Penta, "On the impact of refactoring operations on code quality metrics," in *30th IEEE International*

Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014, 2014, pp. 456–460.

[156] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005*. ACM, 2005.

[157] F. Palomba, G. Bavota, R. Oliveto, F. Fasano, M. Di Penta, and A. De Lucia, "Bad code smells study - online appendix," 2015. [Online]. Available: https://dibt.unimol.it/fpalomba/reports/badSmell-analysis/index.html

[158] M. Lopez and N. Habra, "Relevance of the cyclomatic complexity threshold for the java programming language," *Software Measurement European Forum*, 2015.

[159] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed. Lawrence Earlbaum Associates, 1988.

[160] S. Holm, "A simple sequentially rejective Bonferroni test procedure," *Scandinavian Journal on Statistics*, vol. 6, pp. 65–70, 1979.

[161] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Software Eng.*, vol. 31, no. 4, pp. 340–355, 2005.

[162] D. Hosmer and S. Lemeshow, *Applied Logistic Regression (2nd Edition)*. Wiley, 2000.

[163] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Automated software engineering (ASE), 2013 IEEE/ACM 28th international conference on*. IEEE, 2013, pp. 268–278.

[164] A. Jbara, A. Matan, and D. G. Feitelson, "High-MCC functions in the linux kernel," in *IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13, 2012*, 2012, pp. 83–92.

[165] R. Likert, "A technique for the measurement of attitudes," *Archives of Psychology*, vol. 22, no. 140, 1932.

[166] S. Lemma Abebe, S. Haiduc, P. Tonella, and A. Marcus, "The effect of lexicon bad smells on concept location in source code," in *11th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM 2011*. IEEE, 2011, pp. 125–134.

[167] Y. Baruch, "Response rate in academic studies a comparative analysis," *Human Relations*, pp. 52(4):421–438, 1999.

[168] R. Leitch and E. Stroulia, "Assessing the maintainability benefits of design restructuring using dependency analysis," in *Software Metrics Symposium, 2003. Proceedings. Ninth International*, sept. 2003, pp. 309 – 322.

[169] J. Ratzinger, M. Fischer, and H. Gall, "Improving evolvability through refactoring," in *Proceedings of the 2005 international workshop on Mining software repositories*, ser. MSR '05, 2005, pp. 1–5.

[170] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi, "Does refactoring improve reusability?" in *Proceedings of the 9th international conference on Reuse of Off-the-Shelf Components*, ser. ICSR'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 287–297.

[171] R. Shatnawi and W. Li, "An empirical assessment of refactoring impact on software quality using a hierarchical quality model," *International Journal of Software Engineering and Its Applications*, vol. 5, no. 4, pp. 127–149, 2011.

[172] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*. IEEE Computer Society, 2010, pp. 1–10.

[173] S. Crawford, A. McIntosh, and D. Pregibon, "An analysis of static metrics and faults in c software," *Journal of Systems and Software*, vol. 5, no. 1, pp. 37 – 48, 1985.

[174] V. Basili, L. Briand, and W. Melo, "A validation of object-oriented design metrics as quality indicators," *Software Engineering, IEEE Transactions on*, vol. 22, no. 10, pp. 751–761, Oct 1996.

[175] A. B. Binkley and S. R. Schach, "Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures," in *Proceedings of the 20th International Conference on Software Engineering*, Kyoto, Japan, 1998, pp. 452–455.

[176] L. C. Briand, J. Wuest, and H. Lounis, "Using coupling measurement for impact analysis in object-oriented systems," in *Proceedings of the 15th IEEE International Conference on Software Maintenance*. Oxford, UK: IEEE Press, 1999, pp. 475–482.

[177] L. C. Briand, J. Wüst, S. V. Ikonomovski, and H. Lounis, "Investigating quality factors in object-oriented designs: an industrial case study," in *Proceedings of the 21st International Conference on Software Engineering*. Los Angeles, California, United States: ACM Press, 1999, pp. 345–354.

[178] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering (TSE)*, vol. 31, no. 10, pp. 897–910, 2005.

[179] Y. Liu, D. Poshyvanyk, R. Ferenc, T. Gyimóthy, and N. Chrisochoides, "Modelling class cohesion as mixtures of latent topics," in *Proceedings of the 25th IEEE International Conference on Software Maintenance*. Edmonton, Canada: IEEE Press, 2009, pp. 233–242.

[180] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "An empirical study on the developers' perception of software coupling," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 692–701. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486879

[181] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics*. Prentice-Hall, 1994.

[182] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empirical Software Engineering*, vol. 14, no. 1, pp. 5–32, 2009.

[183] R. M. O'brien, "A caution regarding rules of thumb for variance inflation factors," *Quality & Quantity*, vol. 41, no. 5, pp. 673–690, 2007. [Online]. Available: http://dx.doi.org/10.1007/s11135-006-9018-6

[184] C. Spearman, "The proof and measurement of association between two things," *The American Journal of Psychology*, vol. 100, 1987.

[185] D. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.

[186] T. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 4, pp. 308–320, Dec 1976.

[187] T. Gîrba, S. Ducasse, A. Kuhn, R. Marinescu, and R. Daniel, "Using concept analysis to detect co-change patterns," in *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, ser. IWPSE '07. ACM, 2007, pp. 83–89.

[188] M. L. Collard, H. H. Kagdi, and J. I. Maletic, "An xml-based lightweight c++ fact extractor," in *11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA*. IEEE Computer Society, 2003, pp. 134–143.

[189] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *1996 International Conference on Software Maintenance (ICSM '96), 4-8 November 1996, Monterey, CA, USA, Proceedings*. IEEE Computer Society, 1996, pp. 244–.

[190] R. Agrawal, T. Imielinski, and A. N. Swami, "Mining association rules between sets of items in large databases," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 1993, pp. 207–216.

[191] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, 2004, pp. 563–572.

[192] G. Canfora, L. Cerulo, and M. Di Penta, "On the use of line co-change for identifying crosscutting concern code," in *22nd IEEE International Conference on Software Maintenance (ICSM 2006), 24-27 September 2006, Philadelphia, Pennsylvania, USA*. IEEE Computer Society, 2006, pp. 213–222.

[193] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 574–586, 2004.

[194] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Collard, "Blending conceptual and evolutionary couplings to support change impact analysis in source code," in *Reverse Engineering (WCRE), 2010 17th Working Conference on*, 2010, pp. 119–128.

[195] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "HIST: replication package http://dx.doi.org/10.6084/m9.figshare.1157374," 2014.

[196] M. Girvan and M. E. Newman, "Community structure in social and biological networks." *Proc Natl Acad Sci U S A*, vol. 99, no. 12, pp. 7821–7826, June 2002.

[197] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[198] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, no. 41, pp. 391–407, 1990.

[199] J. K. Cullum and R. A. Willoughby, *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*. Boston: Birkhauser, 1998, vol. 1, ch. Real rectangular matrices.

[200] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Information & Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[201] T. DeMarco, *Structured Analysis and System Specification*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1979.

[202] M. Page-Jones, *The Practical Guide to Structured Systems Design: 2Nd Edition*. Upper Saddle River, NJ, USA: Yourdon Press, 1988.

[203] X. Wang, L. Pollock, and K. Vijay-Shanker, "Automatic segmentation of method code into meaningful blocks to improve readability," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2011, pp. 35–44.

[204] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*, vol. 41, no. 6, pp. 391–407, 1990.

[205] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "Online appendix: A textual-based technique for smell detection," Tech. Rep., http://dx.doi.org/10.6084/m9.figshare.1590962.

[206] F. Palomba, "Textual analysis for code smell detection," in *Proceedings of the International Conference on Software Engineering (ICSE) - Volume 2*. IEEE, 2015, pp. 769–771.

[207] D. Pelleg and A. W. Moore, "X-means: Extending k-means with efficient estimation of the number of clusters," in *Seventeenth International Conference on Machine Learning*. Morgan Kaufmann, 2000, pp. 727–734.

[208] A. Likas, N. Vlassis, and J. J. Verbeek, "The global k-means clustering algorithm," *Pattern Recognition*, vol. 36, no. 2, pp. 451 – 461, 2003, biometrics.

[209] H. Kuusela and P. Paul, "A comparison of concurrent and retrospective verbal protocol analysis," *The American Journal of Psychology*, vol. 113, no. 3, pp. 387–404, 2000.

[210] M. Shaw, "Larger scale systems require higher-level abstractions," *SIGSOFT Softw. Eng. Notes*, vol. 14, no. 3, pp. 143–146, Apr. 1989. [Online]. Available: http://doi.acm.org/10.1145/75200.75222

[211] A. V. Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, Aug 1995.

[212] C. Simons, J. Singer, and D. R. White, *Search-Based Software Engineering: 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings*. Cham: Springer International Publishing, 2015, ch. Search-Based Refactoring: Metrics Are Not Enough, pp. 47–61. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-22183-0_4

[213] I. Candela, G. Bavota, B. Russo, and R. Oliveto, "Using cohesion and coupling for software remodularization: Is it enough?" *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 3, pp. 24:1–24:28, Jun. 2016. [Online]. Available: http://doi.acm.org/10.1145/2928268

[214] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok, "Local vs. global models for effort estimation and defect prediction," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 343–351. [Online]. Available: http://dx.doi.org/10.1109/ASE.2011.6100072

[215] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press,

2013, pp. 712–721. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486882

[216] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 470–481.

[217] Beck, *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[218] A. van Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*, 2001, pp. 92–95.

[219] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 75–84. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.37

[220] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, April 2015, pp. 1–10.

[221] P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[222] G. C. Murphy, "Houston: We are in overload," in *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France*, 2007, p. 1.

[223] G. Fraser and A. Arcuri, "A large scale evaluation of automated unit test generation using evosuite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, p. 8, 2014.

[224] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *Proceedings International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2013, pp. 352–361.

[225] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 107–118. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786838

[226] J. Ferrer, F. Chicano, and E. Alba, "Evolutionary algorithms for the multi-objective test data generation problem," *Softw. Pract. Exper.*, vol. 42, no. 11, pp. 1331–1362, Nov. 2012. [Online]. Available: http://dx.doi.org/10.1002/spe.1135

[227] G. Pinto and S. Vergilio, "A multi-objective genetic algorithm to test data generation," in *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, vol. 1, Oct 2010, pp. 129–134.

[228] K. Lakhotia, M. Harman, and P. McMinn, "A multi-objective approach to search-based test data generation," in *9th Conference on Genetic and Evolutionary Computation*, ser. GECCO '07. ACM, 2007, pp. 1098–1105.

[229] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276–291, Feb. 2013. [Online]. Available: http://dx.doi.org/10.1109/TSE.2012.14

[230] N. Oster and F. Saglietti, "Automatic test data generation by multi-objective optimisation," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, J. Grski, Ed. Springer Berlin Heidelberg, 2006, vol. 4166, pp. 426–438. [Online]. Available: http://dx.doi.org/10.1007/11875567_32

[231] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *IEEE Trans. Software Eng.*, vol. 40, no. 11, pp. 1100–1125, 2014. [Online]. Available: http://dx.doi.org/10.1109/TSE.2014.2342227

[232] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink, "On the interplay between software testing and evolution and its effect on program comprehension," in *Software Evolution*, 2008, pp. 173–202. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-76440-3_8

[233] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, 2011.

[234] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S.-I. Yoo, "The oracle problem in software testing: A survey," 2015.

[235] M. Greiler, A. Zaidman, A. van Deursen, and M. D. Storey, "Strategies for avoiding text fixture smells during software evolution," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 387–396. [Online]. Available: http://dx.doi.org/10.1109/MSR.2013.6624053

[236] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Trans. Software Eng.*, vol. 33, no. 12, pp. 800–817, 2007. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.70745

[237] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *Proceedings of the 22Nd IEEE International Conference on Software Maintenance*, ser. ICSM '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 469–478. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2006.67

[238] B. Ujhazi, R. Ferenc, D. Poshyvanyk, and T. Gyimothy, "New conceptual coupling and cohesion metrics for object-oriented systems," in *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, ser. SCAM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 33–42. [Online]. Available: http://dx.doi.org/10.1109/SCAM.2010.14

[239] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, "Predicting bugs using antipatterns," in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ser. ICSM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 270–279. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2013.38

[240] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, ser. PROMISE '10. New York, NY, USA: ACM, 2010, pp. 9:1–9:10. [Online]. Available: http://doi.acm.org/10.1145/1868328.1868342

[241] R. Bell, T. Ostrand, and E. Weyuker, "The limited impact of individual developer data on software defect prediction," *Empirical Software Engineering*, vol. 18, no. 3, pp. 478–505, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9178-4

[242] D. D. Nucci, F. Palomba, S. Siravo, G. Bavota, R. Oliveto, and A. D. Lucia, "On the role of developer's scattered changes in bug prediction," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, Sept 2015, pp. 241–250.

[243] A. Hindle, "Green mining: A methodology of relating software change and configuration to power consumption," *Empirical Softw. Engg.*, vol. 20, no. 2, pp. 374–409, Apr. 2015. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9276-6

[244] C. Sahin, F. Cayci, I. L. M. Gutiérrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh, "Initial explorations on design pattern energy usage," in *Proceedings of the First International Workshop on Green and Sustainable Software*, ser. GREENS '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 55–61. [Online]. Available: http://dl.acm.org/citation.cfm?id=2663779.2663789

[245] C. Sahin, P. Tornquist, R. Mckenna, Z. Pearson, and J. Clause, "How does code obfuscation impact energy usage?" in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 131–140. [Online]. Available: http://dx.doi.org/10.1109/ICSME.2014.35

[246] J. Reimann, M. Brylski, and U. Amann, "A tool-supported quality smell catalogue for android developers," 2014.

[247] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (e)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 429–440. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.89