Università degli Studi di Salerno

Dipartimento di Informatica

Dottorato di Ricerca in Informatica
XXXIV Ciclo

TESI DI DOTTORATO / PH.D. THESIS

Technical Debt in Software Development: A Multi-Perspective Investigation

FABIANO PECORELLI

SUPERVISOR: **PROF. ANDREA DE LUCIA**

PHD PROGRAM DIRECTOR: **PROF. ANDREA DE LUCIA**

A.A 2020/2021

Al mio nipotino Giovanni.
La più grande gioia della mia vita.

— 20.12.2020 —

ACKNOWLEDGMENTS - RINGRAZIAMENTI

I'm sorry for the English readers but here I'm writing words that come directly from my heart... And my heart speaks Italian.

Quando ho deciso di intraprendere il dottorato di ricerca ero sommerso da mille dubbi: diverse aziende erano pronte ad offrirmi un contratto che mi desse maggiore stabilità, molte delle persone a me vicine mi sconsigliavano di intraprendere una carriera così lunga e complessa, sentivo dire in giro che i dottorandi vengono tutti sfruttati e poi abbandonati a se stessi. Nonostante tutto, sono andato avanti per la mia strada e ad oggi, al termine di questo percorso, posso affermare di aver fatto la scelta giusta.

La mia fortuna più grande è stata quella di far parte di un gruppo di ricerca eccezionale, formato da persone altrettanto eccezionali. Senza di loro molto probabilmente non starei descrivendo un'esperienza così positiva. Ed è proprio a loro che voglio rivolgere i primi e più sentiti ringraziamenti.

Nell'ormai lontano 2016, durante il corso di Ingegneria del Software, ho incontrato per la prima volta Andrea. Dopo solo un mese e mezzo, a metà del corso, con più di un intero semestre davanti e 8 esami ancora da sostenere, gli ho chiesto di farmi da relatore per la Tesi Triennale. In quel momento, inconsapevolmente, stavo avviando un percorso di vita che mi ha condotto fin qui e che spero mi porterà ancora lontano.

Andrea è riuscito a diventare per me un punto di riferimento nel giro di pochissimo tempo. Ho proseguito il mio percorso di studi principalmente grazie a lui ed è ancora grazie a lui se ho deciso di intraprendere la strada del dottorato di ricerca.

Il dottorato sotto la guida di Andrea è un'esperienza per cuori forti. Andrea è una presenza costante che riesce a tenerti sotto pressione anche quando non c'è fisicamente. Ti sprona a dare il 100% anche quando il 10% è sufficiente per arrivare all'obiettivo, a volte anche con modi poco gentili. Perché Andrea è così. Andrea è la classica persona che *"ti vuole bene ma lo dimostra a modo*

suo". Commetti un piccolo sbaglio? Lui riesce a rendertelo estremamente pesante. Fai qualcosa nel modo giusto? Lui ti spiega come avresti potuto farla ancora meglio.

Non nego che, durante questi anni, a volte ho sofferto di periodi estremamente stressanti per questo motivo. Però poi ogni tanto mi fermo e penso che forse è proprio questo che fa la differenza. Gli importanti risultati ottenuti finora non provengono unicamente dai miei sacrifici ma anche, in gran parte, dalla guida ricevuta.

Caro Andrea, molte persone ti stimano semplicemente per il tuo nome, per il tuo enorme peso in ambito accademico e di ricerca. Io ti stimo e ti voglio un gran bene principalmente per la persona che sei, per la tua estrema spontaneità, per il tuo modo di agire sempre nel bene delle persone a cui tieni, per il tuo essere costantemente presente. *GRAZIE!*

Il secondo (solo per seniority) più doveroso e sentito grazie va a Fabio. Fabio ha avuto un impatto fondamentale sulla mia crescita, rappresentando il naturale complemento di Andrea nel guidarmi durante questi anni. Lui è sempre riuscito a parlarmi in termini puramente pratici, come piace a me. Mi ha dato una carica e una motivazione incredibile anche nei momenti più duri, quando pensavo: *"ma chi mo fa fa?"*. Come un leader silenzioso, mi è stato accanto in ogni momento di questo percorso, spesso sacrificando il pochissimo tempo libero a sua disposizione.

Ma Fabio non è soltanto questo. Con lui ho condiviso ogni cosa durante questi anni: notti insonni passate a lavorare, vacanze, partite di calcetto, lunghe chiacchierate. Insomma, è riuscito ad essere allo stesso tempo un grande advisor, un grande amico, e all'occorrenza anche un fratello maggiore.

Palò sono certo che ci sarai sempre per me, così come io per te. Il meglio deve ancora venire.

Un altro enorme grazie va a Dario e Gemma, entrambi elementi imprescindibili di questo mio percorso. Sempre pronti e disponibili a dare consigli e supportarmi in ogni momento.

Grazie alla mia compagna di viaggio, Marianna, per aver condiviso con me ogni momento di questo percorso. Come promesso, abbiamo cominciato e concluso insieme. Ora manca l'ultimo pezzo...

Oltre ai già menzionati Fabio, Dario, Gemma e Marianna, in questi anni ho avuto la fortuna di condividere le mie giornate con persone eccezionali. Grazie a tutti i membri del SeSa Lab per ogni giornata passata insieme, per ogni polletto, per ogni aperitivo. Purtroppo sono andato via sul più bello, quando le cose iniziavano a farsi ancora più interessanti e questo mi lascia un po' di rammarico. So benissimo però che, in un modo o nell'altro, sarò sempre parte di questo fantastico team.

Ringrazio la mia famiglia allargata: quella reale più i miei amici di sempre. Avete sempre sostenuto ogni mia scelta, avete sempre compreso e accettato (anche a malincuore) ogni mio rifiuto, ogni mia indisponibilità, perfino di sera, nei giorni festivi e nei fine settimana. Vi ringrazio soprattutto per avermi sempre tenuto incollato alle mie origini, alle mie abitudini, alle mie passioni. *Crescere è importante, ma farlo rimanendo se stessi lo è ancora di più.*

Una menzione speciale va a mia sorella Luana. I motivi li conosciamo entrambi benissimo. Senza la tua spinta tutto questo non avrebbe mai nemmeno avuto inizio. Grazie infinitamente per tutto quello che hai fatto e che continui a fare per me. Grazie per quelle azioni che fai quasi di nascosto, che spesso passano inosservate agli occhi di tutti ma non ai miei. Grazie per aver dato alla luce Giovanni, regalandomi la più grande gioia della mia vita. La mia prima Tesi l'ho dedicata a te. Questa, la più importante, la dedico a lui che è una parte di te.

Ringrazio infine Francesca per essere stata al mio fianco ogni giorno, per avere ascoltato ogni mio sfogo e avermi continuamente motivato e fatto sentire gratificato. So che in questi anni fin troppo spesso ti sei trovata a dover affrontare situazioni molto più grandi di te. Grazie per essere sempre rimasta. Grazie per aver lottato con le unghie e con i denti per non perdermi. Grazie per avermi insegnato il vero significato del bene incondizionato.

ABSTRACT

Software products need to be constantly maintained and updated to keep being useful and satisfying companies' and users' needs. Developers are often required to perform software maintenance and evolution activities in the shortest possible time in order to make the changes available as soon as possible. As a result, they do not have the possibility to apply ideal development practices, thus introducing the so-called technical debt, i.e., the application of a quick and low-quality solution instead of a better one that would take longer. This will cause a decrease in software quality and require significant maintenance effort in the future.

For this reason, identifying the symptoms of technical debt in advance is of fundamental importance for software companies. However, such symptoms could appear in different forms and at different stages of development, making harder their identification. In the context of this thesis, we face this challenge from several perspectives.

First, we focus on bad code smells, poor design or implementation choices applied in the source code by developers that have been associated with maintainability and understandability degradation. Over the last years, several researchers have been devising tools and techniques for the automatic detection of these design flaws. However, unfortunately, all the proposed detectors appear to be still too limited and inadequate to be applied in real industrial contexts. The first part of this thesis focuses on experimenting with the suitability of machine learning-based code smell detection techniques. Preliminary results demonstrate that machine learning-based techniques still have limited performance for automatic code smell detection, due to several limitations such as (i) the strongly unbalanced nature of the problem, (ii) the subjectivity of the results, and (iii) the limited set of metrics considered so far. This thesis investigates these three limitations separately, proposing specific solutions to overcome them. However, although some advantages have been

reported, machine learning techniques still require more improvement to provide reliable detection of code smells.

Other than studying technical debt in production code, we also consider its presence, as well as its harmfulness, in test code. Testing activities seem to receive way lower attention during software development: tests are often developed without applying proper programming principles or automatically generated with the support of specific tools. Therefore, resulting test suites are often characterized by a low quality that could also reduce their effectiveness in bug discovery. This thesis faces this challenge by presenting a large-scale analysis of test code quality and effectiveness both in traditional systems and in mobile applications in order to understand the test-related factors that are most related to technical issues in production code. The main results confirm that test suites are characterized by a very low code quality and effectiveness, particularly with respect to mobile applications. Moreover, differently from what was previously stated in the literature, some of the quality aspects considered (e.g., size, test smells) have been shown to have a stronger correlation with production code defects as compared to traditional and widely-adopted coverage metrics.

Finally, we also include a discussion on the main lessons learnt and open issues together with some indications about further research directions.

TABLE OF CONTENTS

1

INTRODUCTION

1.1 CONTEXT AND MOTIVATION

Software is eating the world [8]. More and more tasks in our daily lives as well as critical business processes are being taken over by software.

Over the last years, we have been witnessing a rapid digital transformation in a large number of contexts. Just think about all the new digital services that have been implemented, as well as the existing ones that have been adapted to new contexts, due to the COVID-19 pandemic. In just over a year, in-presence activities, such as school, meetings, and conferences, have been moved to virtual platforms; public institutions have started providing new digital services to facilitate remote operations, and so on.

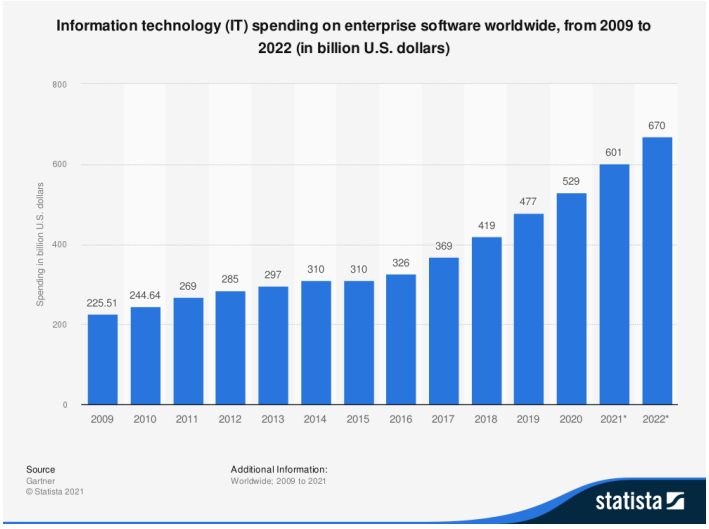

Figure 1.1: Worldwide IT enterprise software spending in the period 2009-2022. Image by Gartner [96].

Figure 1.1 reports information about the amount of money spent per year on enterprise software from 2009 until today. As we can see, over the last ten years the total expenses went from 269 to 601 billions USD, with a further growth of 11% expected for the year 2022.

In such a scenario, software systems, are becoming more and more fundamental for business purposes and for our daily lives. However, at the same time, they are getting larger, more complex, and hard to maintain. Furthermore, the continuous rising of new IT devices brought out the need to constantly adapt software systems to new environments (e.g., big companies are often required to let their software applications work on laptops, smartphones, smartwatches, etc.). As a consequence, software systems are in a continuous and constant change process that makes it difficult for developers to satisfy all requirements in the right way and in the shortest possible time.

A common strategy during software development is to optimize development times, even often neglecting good programming practices and standards. On the one hand, this strategy allows having the software products ready in a short time. On the other hand, such sub-optimal implementations can lead to severe consequences to software quality making it harder to perform maintenance and corrective operations over time.

This phenomenon, is known as technical debt [63], i.e., adopting (intentionally or not) an easy, but limited, solution instead of a better one that would take longer. Identifying and removing technical debt in a short time is a very important activity that has attracted the attention of both practitioners and researchers over the last years. However, it is not always easy to identify the causes leading to technical debt.

Some of the most common causes are:

Market needs, i.e., the urgency of having a product to sell as soon as possible, then released before the necessary changes are complete. This leads to the implementation of quick and inadequate solutions.

Bad implementations of code modules, i.e., when software components are developed ignoring good programming paradigms, leading developers to provide equally poor implementations when adding new code or maintaining the current one.

Absence of proper testing procedures, which encourages "on the fly" bug fixes that do not contemplate possible side effects.

Lack of documentation, where the code is developed "off the cuff", without documentation/specification of requirements. The work to produce the aforementioned documentation a posteriori, and the necessary verification of correspondence with what has already been coded, represents a debt that must be paid sooner or later.

Lack of knowledge, when developers simply do not have the right skills to write good-quality code.

Lack of collaboration, when wrong collaboration/cooperation structures are adopted causing development process efficiency degradation.

This thesis mainly focuses on two of these aspects. First, we focus on the identification of bad code implementations, intended as code smells, poor design or implementation choices applied in the source code by developers [88]. To this aim, we investigate the application of machine learning-based techniques for code smells automatic detection. Then, we move our attention to how developers behave with respect to test code implementation, both in standard and in mobile applications.

1.2 RESEARCH STATEMENT

Over the last decades, many researchers working in the field of software engineering have put great interest in the topics covered in this thesis. However, despite the considerable availability of high-quality contributions in the literature, there are some aspects that have not been properly addressed or can still be improved. As follows, we summarize the main limitations and criticalities identified.

1. **Automatic detection of code smells.** Code smells are one of the main symptoms of technical debt. Several code smell detection techniques have been devised and made available to developers and researchers over time [197, 228, 234, 299]. Most of them rely on heuristic approaches, discriminating code artifacts affected (or not) by a certain type of

code smell through the definition of detection rules that compare the values of relevant metrics against some previously defined thresholds. Despite the accuracy of these approaches has been empirically found to be fairly high, there are some important limitations that threaten the adoption of these heuristic approaches in practice [78, 339]. First, code smells identified by these techniques are subjectively interpreted by developers, meaning that they output code smell candidates that are not considered as actual problems by developers [83, 181]. Furthermore, the agreement between detectors is very low [82], which means that different detectors are required to detect the smelliness of different code components. At last, the performance of most of the current detectors is strongly influenced by the thresholds needed to identify smelly and non-smelly instances [78].

2. **Machine learning for code smell detection.** To overcome heuristic approaches limitations, researchers recently adopted machine learning (ML) to avoid the definition of thresholds and decrease the false positive rate [85]: in this schema, a machine learning classifier is trained on a set of independent variables (a.k.a., predictors) to calculate the value of a dependent variable (i.e., the presence of a smell or the likelihood of a code element to be affected by a smell). Although the use of machine learning looks promising, its actual accuracy for code smell detection is still under debate, as previous work has observed contrasting results [71, 85]. More importantly, it is still unknown whether these techniques actually represent a better solution with respect to traditional heuristic ones. In other words, the problem of assessing the feasibility of machine learning for code smell detection is still open and requires further investigations.

3. **Lack of attention to testing activities.** Software testing is a crucial activity that developers perform to produce high-quality and reliable software. However, it is not properly applied in practice because of two main reasons: (i) Determining the effectiveness of software tests is an open research and practical challenge; (ii) Developers have limited awareness with respect to how much testing they perform and how

much is instead required. As a result, resulting test suites are often hard to understand and maintain as they are implemented without following good programming practices and appear to contain low-quality code. Many researchers in the past have associated some testing aspects to the code quality of the corresponding production code. However, there is still a lack of evidence on what are the test-related factors that really influence software code quality.

4. **Software testing: The mobile applications perspective.** One of the fundamental aspects of mobile applications development is time-to-market. Even more than for standard applications developers, mobile apps developers are always requested to release in the shortest possible time. In such a scenario, code quality is often overshadowed and testing is performed only superficially. Indeed, many bugs in mobile applications are discovered by users when applications are already in production.

This thesis aims to face this criticalities and address the main limitations mentioned above. Specifically, we define four high-level research questions:

- **RQ**$_a$ What are the capabilities of machine learning-based algorithm for code smell detection?
- **RQ**$_b$ How and to what extent can the limitation of machine learning algorithm for code smell detection be overcome?
- **RQ**$_c$ What is the relation of test-related factors on software code quality?
- **RQ**$_d$ Are testing activities performed properly in the context of mobile applications development?

Lower-level research questions are defined in the next chapters.

1.3 RESEARCH CONTRIBUTION

The contribution of this thesis can be divided into two parts. Specifically, we report the research contribution achieved in the context of (i) machine learning-based code smell detection, and (ii) technical debt in test code.

1.3.1 *Research contribution on machine learning-based code smell detection*

To answer our first two high-level research questions, we started comparing the performance of machine learning-based algorithms with the one of heuristic techniques for code smell detection. Preliminary results evidenced that machine learning-based techniques do not outperform heuristic ones. The reason for the low performance seems to be mainly related to three main limitations that we treat separately as follows:

1. **The high data imbalance makes it hard to perform a correct classification.** To overcome this limitation we empirically evaluate the performance variations due to the application of several data balancing techniques, in order to understand whether and to what extent data balancing can improve the performance of machine learning-based code smell detection.

2. **The set of metrics adopted so far does not allow to discriminate smelly and non-smelly instances.** We face this limitation by presenting a novel machine learning-based code smell detection approach that uses static analysis tools' warnings as predictors to classify smelly and non-smelly instances.

3. **Provided predictions are subjectively perceived by developers.** To this aim, we propose a novel developer-driven code smell detection approach. Differently from a standard classification approach, our model tries to rank code smells according to the perceived criticality that developers assign to them.

1.3.2 *Research contribution on technical debt in test code*

To answer $\mathbf{RQ}_c$ and $\mathbf{RQ}_d$, we conduct a multivocal literature review (MLR) with the aim of collecting all the test-related factors that have been associated to software code quality in the past. Then, we provide a large empirical study to find statistical evidence of this relation, both in standard and in mobile applications. Results of our study evidence that not only coverage-related metrics relate to software code quality but also other intrinsic characteristics (e.g., test case size, presence of test smells).

1.4 STRUCTURE OF THE THESIS

The remainder of this thesis is organized into three parts.
Part I groups the studies on machine learning for code smell detection:

- Chapter 2 provides a background and an analysis of the current state of the art on code smell detection;
- Chapter 3 describes a comparative study we conducted between heuristic and machine learning-based techniques for code smell detection;
- Chapter 4 provides a deep analysis of the application of data balancing techniques for code smell detection;
- Chapter 5 presents a machine learning-based code smell detection approach built on top of the warnings generated by static analysis tools;
- Chapter 6 presents a novel code smell prioritization approach based on the way developers perceive the criticality of code smells in source code;
- Chapter 7 discusses the main results in response to the high-level research questions and reports all the major threats to validity.

Part II describes the studies related to technical debt from a testing perspective:

- Chapter 8 provides a background and an analysis of the related literature on technical debt from a testing perspective;
- Chapter 9 reports a multivocal literature review aiming to extract all the test-related factors related to software code quality;
- Chapter 10 reports an empirical investigation on the relation of test-related factors to software code quality;
- Chapter 11 reports a large empirical investigation on the presence, quality and effectiveness of test suites in android mobile applications;
- Chapter 12 discusses the achieved results in response to RQ_c and RQ_d and reports all the major threats to validity.

Part III concludes the thesis and discusses the future directions and challenges in technical debt research:

- Chapter 13 reports a summary of the work presented in this thesis and discusses the main lesson learnt and open issues that need to be addressed in the future other than delineating the future research directions and reporting details about some preliminary analyses already carried out.

Part I

MACHINE LEARNING FOR CODE SMELL DETECTION

2

BACKGROUND & RELATED WORK

2.1 INTRODUCTION, MOTIVATION, AND RELATED WORK

One of the foremost indications of the presence of technical debt is represented by *code smells* [88], i.e., sub-optimal design solutions that developers apply on a software system. Long methods implementing several functionalities, classes having complex structures, or excessive coupling between classes are just few examples of code smells typically observable in existing software systems [226].

In recent years, code smells have been investigated under different perspectives [15, 249]. Their introduction [302, 304] and evolution [14, 47, 219, 225, 252], their impact on reliability [239, 240] and maintainability [137, 226], as well as the way developers perceive them [227, 288, 329] have been deeply analyzed in literature and have revealed that code smells represent serious threats to source code maintenance and evolution. Most notably, the impact of code smells on program comprehension has been investigated by Abbes *et al.*[1] and Yamashita and Moonen [331]. Both studies have demonstrated that code smells negatively impact program comprehension by reducing the maintainability of the affected classes.

For all these reasons, several techniques to automatically identify code smells in source code have been widely investigated [78, 229]. Most of these techniques rely on heuristics and discriminate code artefacts affected (or not) by a certain type of smell through the application of detection rules that compare the values of relevant metrics extracted from source code against some empirically identified thresholds. As an example, Moha *et al.*[197] introduced DECOR, a method to specify and detect code and design smells using a Domain-Specific Language (DSL). Following the general process

described above, DECOR uses a set of rules, called "rule card"[1], that describe the intrinsic characteristics of a class affected by a smell. For instance, a *Blob* is detected when a class has an LCOM5 (Lack of Cohesion Of Methods) [121] 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. The authors showed that DECOR can identify smells with an average F-Measure of ≈80%.

Marinescu [186] 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 [157] 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.

Tsantalis *et al.*[300] presented JDEODORANT, a tool whose first version was able to detect *Feature Envy* bad smells and suggest move method refactoring opportunities. Afterwards, other code smells have been supported (i.e., *State Checking*, *Long Method*, and *Blob*) [81, 299, 301]. The detection strategies for these smells are based on code metrics that are then connected to each other using supervised clustering algorithms and thresholds to cut the resulting dendrograms. The empirical assessment of the performance of JDEODORANT showed its high accuracy (on average, ≈75%).

Bavota et al. [20] proposed the use of structural and conceptual analysis to support the detection of God Classes through the identification of Extract

1 http://www.ptidej.net/research/designsmells/

Class Refactoring opportunities. In particular, a class of the system under analysis is first parsed to build a method-by-method matrix. A generic entry ci,j of the method-by-method matrix represents the likelihood that method mi and method mj 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) [109], the Call-based Dependence between Methods (CDM) [21], and the Conceptual Similarity between Methods (CSM) [184]. 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).

Similarly, Tsantalis and Chatzigeorgiou [301] proposed a technique to detect Long Method code smell instances throug the identification of Extract Method Refactoring opportunities. Specifically, the technique employs a block-based slicing technique [189] 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.

Tsantalis et al. [300] also devised a technique to detect Feature Envy instances by identifying Move Method Refactoring opportuniites. This technique 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 [24], 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) [285]. 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).

Bavota et al. [23] 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 [211] defines the compromise between coupling and cohesion.

Simon et al. [275] 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.

Munro [203] 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 [309] 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. [264] 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.

Palomba *et al.*[228] presented HIST, an approach to detected code smells by using source code evolution information. The method they propose extract information about how the code has changed over a period of time. This information is used by a detection model to detect code smells in the source code.

Palomba *et al.*[234] also presented TACO (Textual Analysis for Code smell detectiOn). TACO follows a three-step process: (i) first it extracts the textual content of the component under analysus, (ii) then it applies Information Retrieval (IR) normalization process, and (iii) finally it performs code smell detection based on textual information and similarities by relying on specific heuristics.

Despite the good performance achievable with the discussed techniques, previous work [78, 339] pointed out three important limitations that might preclude their use in practice: (i) subjectiveness of developers with respect to code smells detected by these tools, (ii) scarce agreement between different detectors, and (iii) difficulties in finding good thresholds to be used for detection. The adoption of machine learning techniques may potentially mitigate these problems, however there is limited evidence of whether and how much machine learning actually improves the performance of traditional approaches.

Nevertheless, there are some important limitations that threaten the adoption of these heuristic approaches in practice [78, 339]. First, code smells identified by these techniques are subjectively interpreted by developers, meaning that they output code smell candidates that are not considered as actual problems by developers [83, 181]. Furthermore, the agreement between detectors is very low [82], which means that different detectors are required to detect the smelliness of different code components. At last, the performance of most of the current detectors is strongly influenced by the thresholds needed to identify smelly and non-smelly instances [78].

To overcome these limitations, researchers recently adopted machine learning (ML) to avoid thresholds and decrease the false positive rate [85]: in this schema, a classifier (e.g., Logistic Regression [6]) exploits a set of independent variables (a.k.a., predictors) to calculate the value of a dependent

variable (i.e., the presence of a smell or degree of the smelliness of a code element).

In this context, Kreimer [151] proposed a prediction model that, on the basis of code metrics used as independent variables, can lead to high values of accuracy. It adopts DECISION TREES to detect two code smells (i.e., *Blob* and *Long Method*). Later on, Amorim *et al.*[7] confirmed the previous findings by evaluating the performance of DECISION TREES on four medium-scale open-source projects. Vaucher *et al.*[316] studied *Blob*'s evolution relying on a NAIVE BAYES classifier, whereas Maiga *et al.*[177, 178] proposed the use of SUPPORT VECTOR MACHINE (SVM) and showed that such a model can reach an F-Measure of ≈80%. The use of BAYESIAN BELIEF NETWORKS to detect *Blob*, *Functional Decomposition*, and *Spaghetti Code* instances on open-source programs, proposed by Khomh *et al.*[138, 139] lead to an overall F-Measure close to 60%. Similarly, Hassaine *et al.*[118] defined an immune-inspired approach for the detection of *Blob* smells, while Oliveto *et al.*[221] used a B-Splines to detect them. More recently, some authors investigated the feasibility of machine learning to detect code clones [318, 323, 332]. Arcelli Fontana *et al.*made the most relevant progress in this field [85–87]. In their work, they (i) theorised that ML might lead to a more objective evaluation of code smells hazardousness [87], (ii) provided a ML method to assess code smell intensity [86], and (iii) compared 16 ML techniques for the detection of four code smell types [85] showing that ML can lead to F-Measure values close to 100%. Nevertheless, recently Di Nucci *et al.*[71] demonstrated that, in a real use-case scenario, the results achieved by Arcelli Fontana *et al.*[85] cannot be generalised, thus contrasting the real effectiveness of machine learning for code smell detection.

Although the use of machine learning looks promising, its actual accuracy for code smell detection is still under debate, as previous work has observed contrasting results [71, 85]. More importantly, it is still unknown whether these techniques actually represent a better solution with respect to traditional heuristics. In other words, the problem of assessing the feasibility of machine learning for code smell detection is still open and requires further investi-

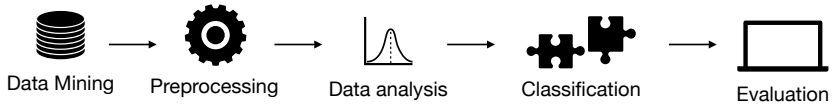

Figure 2.1: Machine learning process for code smell detection.

gations. The next section reports details about the application of machine learning techniques for code smell detection.

2.2 BACKGROUND

Figure 2.1 depicts the pipeline of a standard machine learning process for code smell detection. Such a process, is generally characterized by 5 steps as shown in the figure:

Step 1 - Data mining. This step consists of mining data from software systems' repositories in order to obtain all the information needed to train/test machine learning classifiers. As the main outcome, this phase provides a data set reporting the values of all the mined metrics for each of the code components under analysis.

Step 2 - Preprocessing. Once mined all the necessary data, some preprocessing steps are required to correct/remove possible biases contained in the data. Common practices performed during this phase are the application of data imputation techniques [307] to manage eventually missing values or the evaluation of the multicollinearity between the mined variables in order to avoid biased results. The most important aspect to deal with during preprocessing is how to assign values to the target attribute, i.e., the attribute that we are trying to predict. In code smell detection, usually, the target attribute is represented by a binary value indicating if the specific code component under consideration is affected or not by a code smell type. To assign values to such an attribute, generally, a manual validation is performed, i.e., some code inspectors are requested to read the source code and manually label the presence/absence of code smells for all the components of a system.

Step 3 - Data analysis. After the first two phases, we should now have a curated data set containing both input and target attributes which can be further analyzed and optimized. Two optimizations that are fundamental when dealing with code smell detection are feature selection and data balancing. Feature selection consists of extracting from the input variables the most powerful predictors of the target attribute. Some of the most common feature selection techniques that we have used in the studies presented in this thesis are (i) the Correlation-based Feature Selection (CFS) approach [113], which uses correlation measures and a heuristic search strategy to identify a subset of actually relevant features for a model, or (ii) the Gain Ratio Feature Evaluation technique [257], that establishes a ranking of the features according to their importance for the predictions done by the different models. Given a set of features F = $\{f_1, ..., f_n\}$ belonging to the model M, the *Gain Ratio Feature Evaluation* computes the difference, in terms of Shannon entropy, between the model including the feature f_i and the model that does not include f_i as independent variable. The higher the difference obtained by a feature f_i, the higher its value for the model. The outcome is represented by a ranked list, where the features providing the highest gain are put at the top.

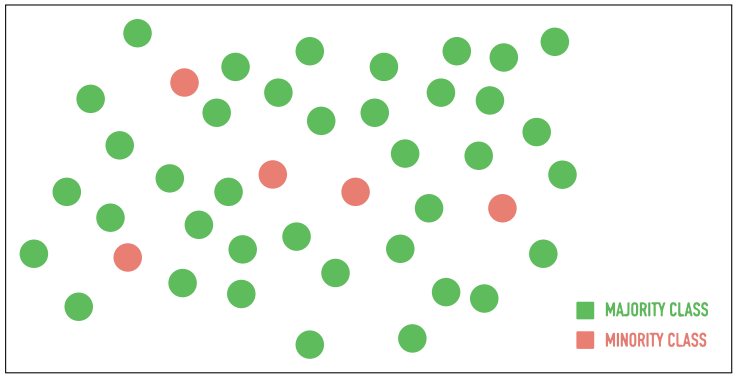

Figure 2.2: An example of unbalanced dataset

Another aspect to deal with in the data analysis step is data balancing. In the case of code smell detection, the distribution of the target attribute is not uniform, for instance, the target attribute could be 'true' for 5% of the dataset, while it is 'false' for the remaining 95%. If the training set does

not contain enough examples for all the values of a certain target attribute, the ML model will not learn how to distinguishes it. In these cases, data balancing techniques can solve the problem and allow the learner to get enough examples to be trained. Figure 2.2 plots a simplified representation of an imbalanced dataset in which most of the instances belong to the *green* majority class. The descriptions of the data balancing techniques used in this thesis are reported below:

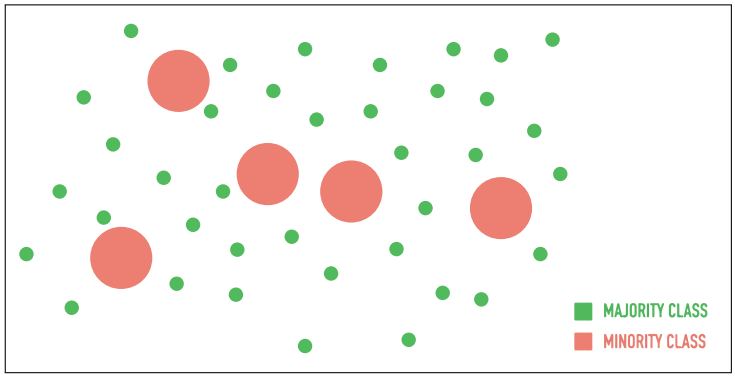

Figure 2.3: Example of application of Oversampling

Oversampling [168]. This algorithm randomly adds samples of the minority class. Figure 2.3 shows a representation of the effects of the algorithm. In this representation, instances are not added or removed, but their weights are modified in such a way that more importance is given to the instances belonging to the minority class.

Undersampling. This algorithm randomly removes samples of the majority class using either sampling with or without replacement. A common practice is to replace instances of the majority class (i.e., clean classes) with instances from the minority class (i.e., smelly classes) until obtaining an even number of instances for both classes [72, 92] as shown in Figure 2.4. Please notice that in the figure, the size of a point represents its frequency.

Synthetic Minority Oversampling TEchnique [48]. This technique increases the number of instances from the minority class by generating new synthetic instances based on the nearest neighbours belonging to that class. As shown

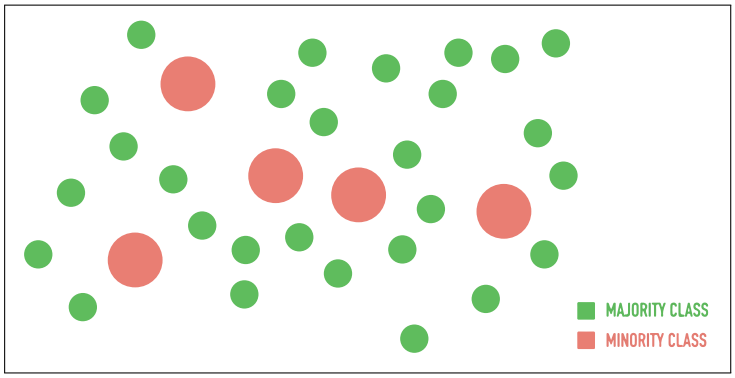

Figure 2.4: Example of application of Undersampling

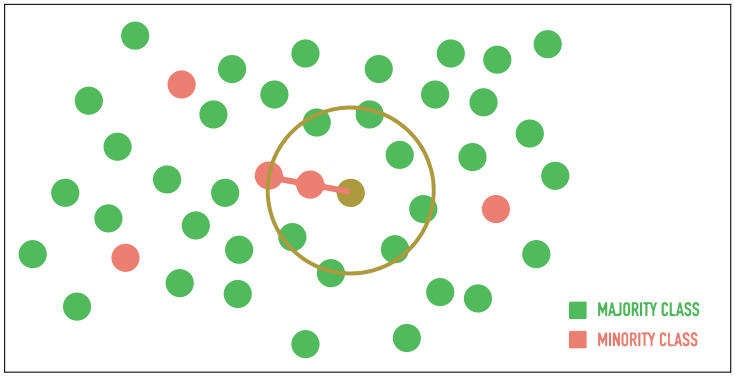

Figure 2.5: Example of application of SMOTE

in Figure 2.5, to create a new synthetic instance, *SMOTE* randomly selects an element from the minority class and identifies its nearest neighbours: the new instance is created between them. The number of nearest neighbours to use is a parameter of the algorithm.

Cost-Sensitive Classifier [150]. A Cost-Sensitive Classifier is a meta-classifier that renders a cost-sensitive version of the base classifier. The training instances can be re-weighted according to the total cost assigned to each class, i.e., the cost-sensitivity is considered during the training phase. Considering that ML-based code smell detection exhibits many false negatives, we configure the CostSensitiveClassifier provided by Weka [114] in such a way that the cost of false negatives is twice the cost of false positives.

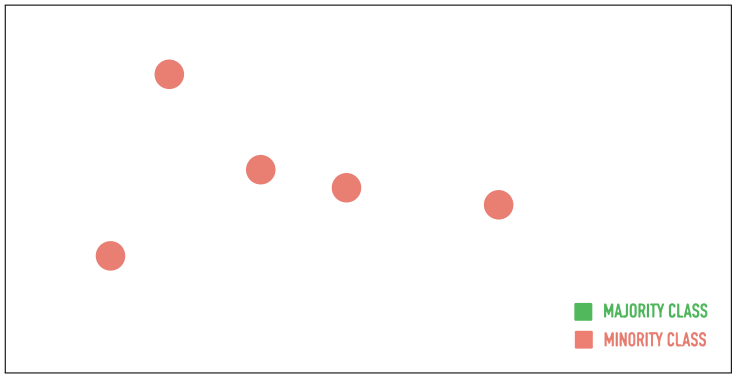

Figure 2.6: Example of application of One-Class Classifier

One-Class Classifier [297]. As shown in Figure 2.6, a One-Class Classifier is trained only on the samples belonging to the minority class to learn the unique features of this class and accurately identify an unseen sample of this class as distinct from a sample of any other class. All instances belonging to other classes are identified as outliers.

Step 4 - Classification. After having applied all the data optimizations, we are now ready to perform the classification. In this phase, the input data are split into two sets: the training set, which contains all instances used to train the classifier, and the test set, which contains the instances for which we want to perform the classification. Then, a machine learning classifier (e.g., Naive Bayes, Random Forest) is trained on the training set data in order to find patterns between all the input variables and the target variable, which is known within the training set. Once the classifier has found such patterns, it performs a classification to predict the values of the target attribute for all the instances contained in the test set.

Note that some machine learning classifiers require the specification of hyper-parameters to be executed properly. Despite they usually rely on a the default configuration, their performance may lose up to 30% of their classification capabilities. As such, tuning hyper-parameters is key to obtain a more accurate model. The available toolkits provide various configuration algorithms. Example of configuration algorithms are: (i) GridSearch, which exhaustively verifies how variations of hyper-parameters impacts the perfor-

mance of the model, so that the best configuration can be found, and (ii) MULTISEARCH [334], which implements a multidimensional search of the hyper-parameter space to identify the best configuration of the model based on the input data.

Step 4 - Evaluation. The last step consists in testing the performance of the adopted machine learning technique. Depending on the problem domain and specification, various methodologies are available:

Percentage Split. The idea is to split the labeled dataset in two parts. The first is then used to train the machine learner, while the latter will be the test set. In this way, you can measure how well the model predicts the instances of the test set.

K-fold Cross Validation. The dataset is randomly partitioned in k folds (usually ten). K-1 of them are then used as training set, while one is retained as test set. The process is then repeated k times, so that each fold is used as test once.

Leave One Out Cross Validation. Similar to k-fold cross validation, but here each fold contains only one instance. Therefore, iteratively, all the remaining N-1 instances are used as training data to predict the value of a single instance in the test set. One known and widely used variant of this validation strategy is the leave on group out cross validation (LOGO), where each fold contains data belonging to a different group. For instance, we could assign different groups to data coming from different software projects, then we use data coming from external projects to train a classifier that is tested on the project we are interested in.

Finally, to actually assess the performance of the machine learning model we need to measure some evaluation metrics. The definitions of the most used evaluation metrics are reported below.

Let *TP* (True Positives) be the actual smelly instances that have been correctly identified as smelly by a model, *FP* (False Positives) the non-smelly instances that have been erroneously identified as smelly, *TN* (True Negatives) the non-smelly instances that have been correctly identified as non-smelly, and *FN* (False Negatives) the smelly instances that have been erroneously identified as non-smelly, we can measure:

- **Accuracy.** This metric represents the fraction of instances that are correctly classified:

$$Accuracy = \frac{\#TP + TN}{\#(TP + FP + TN + FN)}\% \tag{2.1}$$

- **Precision.** This metric represents the fraction of instances predicted as smelly that are actually smelly, namely:

$$Precision = \frac{\#TP}{\#(TP + FP)}\% \tag{2.2}$$

- **Recall.** This metric represents the fraction of actually smelly instances that have been correctly predicted as smelly:

$$Recall = \frac{\#TP}{\#(TP + FN)}\% \tag{2.3}$$

- **F-measure.** This metric is defined as the weighted harmonic mean of the precision and recall, and it is computed as:

$$F - Measure = 2 * \frac{precision * recall}{precision + recall}\% \tag{2.4}$$

- **MCC.** MCC is a correlation coefficient between the observed and predicted binary classifications. It has values in the range [-1,+1] where a coefficient of +1 represents a perfect prediction and 1 indicates total disagreement between prediction and observation:

$$MCC = \frac{\#(TP*TN - FP*FN)}{\#\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \tag{2.5}$$

Table 2.1: Descriptions of the code smells considered in the thesis

Code Smell	Description
God Class	This smell characterises classes having a large size, poor cohesion, and several dependencies on other data classes of the system [88]
Spaghetti Code	Classes affected by this smell declare several long methods without parameters [88]
Class Data Should Be Private	This smell appears in cases where a class exposes its attributes, thus violating the information hiding principle [88]
Complex Class	Classes presenting an overly high cyclomatic complexity [191] are affected by this design flaw
Long Method	Methods implementing more than one functionality are affected by this smell [88]
Feature Envy	This smell arises when a method communicates more with methods that are inside another class than the ones in its
Inappropriate Intimacy	This smell occurs when two classes are highly coupled [197, 226]
Lazy Class	This smell targets classes that do not have enough responsibilities within the system and that, therefore, should be removed to reduce the overall maintainability costs [88]
Middle Man	This smell arises when a class delegates to other classes most of the methods it implements [88]
Refused Bequest	A class which redefines most of its inherited methods, then making the hierarchy wrong [88]
Speculative Generality	This smell shows up when a class declared as abstract has very few children using its methods [88]
Long Parameter List	A method having a long list of parameters is harder to use [88]
Shotgun Surgery	This smell arises when a change to a class (e.g., to one of its fields/methods) triggers many little changes to other classes of the system [88]
Brain Repository	Repository classes that include complex business logic or queries [10]
Fat Repository	A Repository which deals with many Entity classes [10]
Promiscuous Controller	Controller classes exhibiting this smell offer too many actions [10]
Brain Controller	Controller classes with a complex control flow [10]

2.3 OUR CONTRIBUTION ON ML-BASED FOR CODE SMELL DETECTION

This part of the thesis aims at performing steps ahead toward the application of machine learning for code smell detection. To this aim we conducted empirical studies and experimented new techniques for the automatic detection of different code smell types. Descriptions of the code smells considered in this thesis are reported in Table 2.1.

First of all, we started addressing our first high-level research question (i.e., $\mathbf{RQ}_a$) in order to provide a deeper knowledge on the capabilities of machine learning for code smell detection.

To this aim, in Chapter 3 we conduct an empirical study comparing the performance of machine learning-based and heuristic techniques for code smell detection.

Our main findings report that heuristic techniques have slightly better performance than machine learning approaches, thus indicating that machine learning-based techniques for code smell detection still need further improvement to be applied in practice. In particular, we identified three major limitations of these approaches that could strongly downsizing their performance: (i) high data imbalance can lead machine learning algorithm to misclassifications, (ii) the set of metrics adopted so far is still to limited, and (iii) the outcoming results are subjectively perceived by developers. Therefore, we moved our attention to the second high-level research question (i.e., $\mathbf{RQ}_b$) to understand whether and to what extent these limitations can be overcome.

First we focus on the data imbalance limitation. As code smell detection is a problem in which training datasets usually have skewed class proportions (i.e., highly imbalanced data) [71], data balancing is a key factor to improve the reliability of such models. We face this issue in Chapter 4 by presenting a comprehensive comparison of several data balancing techniques for the automatic detection of Object-Oriented (OO) and Model-View-Controller (MVC) code smells.

After having analyzed the data imbalance limitation, we consider the exploration of new metrics as code smell predictors. In particular, in Chapter 5 we present a novel machine learning-based technique that relies on warnings generated by by three static analysis tools, i.e., CHECKSTYLE, FINDBUGS, and PMD, as predictors for code smell detection. The choice of focusing on those warnings was motivated by the type of design issues that can be identified through static analysis tools. More particularly, while some of the warnings they raise are not directly related to source code design and code quality, there are several exceptions. For instance, let consider the warning category called *'bad_practice'* raised by FINDBUGS, one of the most widely

used static analysis tools in practice [315]. According to the list of warnings reported in the official documentation,[2] this category includes a number of design-related warnings. Similarly, the warning category *'design'* provided by CHECKSTYLE and PMD is also associated with design issues. As such, static analysis tools actually deal with the design of source code and pinpoint a number of violations that may be connected to the presence of code smells. In the context of this chapter, we first hypothesized that the indications provided by the static analysis tools [320] can be potentially useful to characterize code smell instances. Secondly, we conjectured that the incorporation of these warnings within intelligent systems may represent a way to reduce the high amount of false positives they output [127].

Finally, in Chapter 6, we focus on the third (and last) identified limitation, i.e., the developers' subjective interpretation of the results provided by machine learning-based code smell detection techniques. To overcome this limitation, we change the way to detect code smells in source code components. In particular, instead of performing a binary classification to predict the presence/absence of code smells based on a manual validated dataset, we propose a novel technique that aims to predict the developers perceived criticality of code smells in source code based on a dataset composed of developers' perception of the severity of 1,332 code smell instances collected through a survey.

2 The FINDBUGS official documentation: http://findbugs.sourceforge.net/bugDescriptions.html.

3

COMPARING HEURISTIC AND MACHINE LEARNING APPROACHES FOR METRIC-BASED CODE SMELL DETECTION

This chapter presents a large-scale empirical study—that features 125 releases, 13 software systems, and 5 code smell types—in which we compare the performance of machine learning techniques and heuristic approaches for code smell detection. We experiment with five code smell detection models built using different algorithms and compare their performance with DECOR [197], a state-of-the-art heuristic-based approach that is the most adopted one in literature [78, 249].

The main contributions of this chapter can be summarized as follow:

1. A large-scale empirical comparison of machine learning and heuristic approaches for metric-based code smell detection, which highlighted that heuristic approaches still perform better, yet have low performance.

2. The identification of limitations, such as data unbalancing or poor precision, that make the application of both machine learning models and heuristic approaches for code smell detection hard.

3.1 EMPIRICAL STUDY DEFINITION AND DESIGN

The *goal* of the study is to compare heuristic with machine learning approaches for code smell detection, with the *purpose* of assessing the extent to which code smell detection models can be effectively used in practice. The *perspective* is of both researchers and practitioners: both of them are interested in evaluating the performance of machine learning for code smell detection, while the former are interested also in understanding possible limitations of the current

approaches. More specifically, we aim at addressing the following research question:

RQ$_1$. *How do machine learning-based techniques for code smell detection perform when compared to a baseline heuristic-based approach?*

The following sections report the methodological steps that we conducted to address **RQ**$_1$.

3.1.1 *Context of the Study*

The *context* of the study was represented by software systems and code smells. As for the former, we exploited a publicly available dataset composed of 125 releases of 13 open source software systems [226], whose description is reported in Table 3.1. The projects of the dataset are heterogeneous, have different size, lifetime, and belong to various application domains: as such, we could reduce threats to the generalizability of the empirical study. The dataset contains a set of $8,534$ *manually validated* code smell instances of five different types: thus, we could perform our study on a dataset of real code smells, in contrast with the artificially created ones that were used in previous research on code smell detection [15].

Table 3.1: Projects Considered in the Study

Project	Description	# Releases	# Classes	# Methods
Ant	Build System	10	1,002-1,218	9,333-11,919
ArgoUML	UML Modeling Tool	13	889-2,221	7,252-17,309
Cassandra	Database Management System	8	470-727	4,422-7,901
Derby	Relational Database Management System	9	1,733-2,920	23,107-421,183
Eclipse	Integrated Development Environment	21	812-5,736	10,819-51,008
Elastic Search	RESTful Search and Analytics Engine	8	1,785-2,466	12,393-18,225
Hadoop	Tool for Distributed Computing	9	148-344	1,224-3,080
HSQLDB	HyperSQL Database Engine	10	556-601	10,075-11,016
Incubating	Codebase	6	497-787	4,210-5,767
Nutch	Web-search Software	7	304-453	1,846-2,761
Qpid	Messaging Tool	5	1,547-2,118	14,858-20,402
Wicket	Java Application Framework	9	2,133-2,212	12,370-12,824
Xerces	XML Parser	10	483-542	5,280-6,126

Table 3.2: Descriptive statistics for smells distribution

Code Smell	min	mean	median	max	total
God Class	0	5.5	4	24	509
Spaghetti Code	0	12.7	11	31	1443
Class Data Should Be Private	0	11.4	11	37	1150
Complex Class	0	6.4	4	20	669
Long Method	3	48.3	26	147	4763

With respect to code smells, we considered five different types defined in the catalog by Fowler [88], namely *God Class*, *Spaghetti Code*, *Class Data Should be Private*, *Complex Class*, and *Long Method*. Detailed descriptions of code smells are reported in Table 2.1.

Table 3.2 reports the descriptive statistics related to the distribution of code smells over the considered dataset. As it is possible to observe, the median number of code smells in each considered release is pretty low (it ranges from 4 to 26). The absolute numbers correspond to extremely low relative percentages: as an example, we noticed that the maximum number of God Class instances (24), in the project APACHE DERBY 10.3.3.0, only represents the 1% of the total number of classes belonging to this system (2220). On the one hand, the observed distribution confirms previous findings in the field [226]. On the other hand, the extremely low number of code smells clearly evidences that the problem in question is highly unbalanced.

In the remaining of this section, we explain the machine learning-based and heuristic-based detection solutions exploited in our study to identify instances of these code smells.

3.1.2 *Heuristic-Based Detection of Code Smells*

Among all the techniques and tools available for code smell detection [78, 249], we relied on DECOR [197] as a metric-based heuristic baseline for several reasons. First, this technique has been extensively used and showed good detection performance (e.g., [138, 139, 228, 234]), thus representing a valid candidate to be a baseline in our study. Furthermore, it is based on

detection rules that can be computed directly looking at the source code of a class/method, without the need of computationally-expensive operations (e.g., the construction of the Abstract-Syntax Tree and the subsequent clustering mechanism applied by JDeodorant [300]) that would have made the detection phase unfeasible because of the amount of data we had to analyze. Last but not least, DECOR is publicly available. It provided out-of-the-box the detection rules able to identify two code smells considered in the study (i.e., *God Class* and *Spaghetti Code*), while for the remaining ones we relied on the definitions provided by Tufano et al. [304]

God Class. A smelly instance is detected when a class has a size higher than 500 lines of code and either an LCOM5 (Lack of Cohesion Of Methods) [121] higher than 20 or a number of methods and attributes higher than 20.

Spaghetti Code. Decor identifies this smell in cases where a class has a size higher than 600 lines of code and a number of long methods (identified as explained later) without parameters higher than 0.

Class Data Should Be Private. In this case, Decor computes the Number Of Public Attributes (NOPA) of a class and, if this is higher than 10, then a smell is identified.

Complex Class. The detection rule for this smell considers the Weighted Methods per Class (WMC) metric, namely the sum of the McCabe's cyclomatic complexity [191] of the methods of a class. If WMC is higher than 50, a class is detected as affected by the smell.

Long Method. To detect this smell, the lines of code of the method (LOC_METHOD) and the number of parameters of the method (NP) are used. Decor indicates the presence of the smell is a method has more than 100 lines of code and at least one parameter.

Table 3.3 reports the summary of all the detection rules. The full name of the metrics is reported in Table 3.4. We ran Decor over all the 125 releases

Table 3.3: Detection Rules Used by the Heuristic-Based Approach.

Code Smell	Detection Rule
God Class	ELOC > 500 ∧ (NOM+NOA > 20 ∨ LCOM > 20)
Spaghetti Code	ELOC > 600 ∧ NMNOPARAM > 0
Class Data Should Be Private	NOPA > 10
Complex Class	WMC > 50
Long Method	LOC_METHOD > 100 ∧ NP > 1

Table 3.4: Full Names of the Considered Metrics.

Acronym	Full Name
ELOC	Effective Lines Of Code
LCOM	Lack of COhesion in Methods
LOC_METHOD	Lines Of Code of METHOD
NOA	Number Of Attributes
NOM	Number Of Methods
NOPA	Number Of Public Attributes
NP	Number of Parameters
NMNOPARAM	Number of Methods with NO PARAMeters
WMC	Weighted Methods Count

and, on the basis of the output recommendations, we re-constructed the confusion matrices. These were analyzed and compared with those obtained with the machine learning models in terms of the evaluation metrics described in Section 3.1.4.

3.1.3 *Machine Learning-Based Detection of Code Smells*

Once we had defined the heuristic-based baseline, we configured the machine learning-based classifiers to detect the considered smells. This required several steps, ranging from the definition of the dependent and the independent variables to the pre-processing actions needed to avoid common problems such as multi-collinearity and biased interpretation [217].

Dependent variable. Since in our work we were interested in detecting code smells, we set the presence/absence of a certain code smell as

dependent variable of the machine learning model. This information was already available in the considered dataset.

Independent variables. As the overall goal of the study was the comparison between heuristic and machine learning-based detectors, we wanted to avoid that such a comparison could have been biased by other co-factors. For this reason, the independent variables of the model were exactly the same used by the heuristic approach (see Table 3.3): in this way, we avoid the possibility that different performance might be due to the selected metrics rather than the technique exploited.

Selection of the classifier. While several classifiers have been previously used for code smell detection, the related literature showed that it is unclear which of them represents the best solution [15]. For this reason, in this work we compared the five most commonly used ML algorithms such as J48 [258], RANDOM FOREST [32], NAIVE BAYES [126], SUPPORT VECTOR MACHINES [46], and JRIP [58]. To perform a fair comparison, we applied the same configuration, preprocessing, and training strategies to all the classifiers, as described in the following.

Configuration and preprocessing steps. Before assessing the accuracy of the machine learning-based models, we took into account two aspects, i.e., classifier configuration and feature selection, that might possibly bias their performance [176, 296, 328]. We configured the hyper-parameters of the considered classifiers by exploiting the GRID SEARCH algorithm [29]. Secondly, we removed highly correlated independent variables through the CORRELATION-BASED FEATURE SELECTION (CFS) approach [113]. We applied the feature selection algorithm on each release independently, so that the model took into account only the features that are relevant for a specific release of the considered systems. It is important to note that we consciously avoid the application of balancing algorithms [48], i.e., techniques that ensure a similar proportion of smelly and non-smelly classes/methods in the training set. This decision was taken as a result of experimental tests,

Table 3.5: Aggregate Results for Precision, Recall, F-Measure, and MCC

	Precision		Recall		F-Measure		MCC	
	NB	DECOR	NB	DECOR	NB	DECOR	NB	DECOR
God Class	**0.27**	0.08	0.85	**1**	**0.41**	0.16	**0.47**	0.28
Spaghetti Code	**0.15**	0.11	0.30	**0.47**	**0.20**	0.18	0.20	**0.22**
Class Data Should Be Private	**0.29**	0.23	0.34	**0.42**	0.29	**0.30**	0.29	**0.31**
Complex Class	0.23	0.23	0.57	**0.72**	0.33	**0.35**	0.36	**0.37**
Long Method	0.15	**0.57**	**0.56**	0.37	0.23	**0.44**	0.30	**0.42**

where we observed that such algorithms can bias the interpretation of the results in the context of code smell detection.

Validation strategy. To assess the capabilities of the machine learning models, we adopted 10-*Fold Cross Validation* [286]. The process was repeated 10 times, using each time a different fold as test set.

The result of the process described above consisted of a confusion matrix for each code smell type, for each of the 125 releases of the considered projects and for each experimented classifier. These matrices have been later analyzed to measure the evaluation metrics described in the following section.

3.1.4 *Data Analysis and Metrics*

To assess the performance of the experimented detection techniques we computed three well-known metrics [16], namely, *precision*, *recall* and *F-measure*. In addition, we also computed the *Matthews Correlation Coefficient (MCC)* [256]. Since we considered several releases of several systems, we needed to aggregate the results achieved for each release to have a clearer overview of the performance [12]. Therefore, we aggregated the obtained confusion matrices before computing precision, recall, F-Measure, and MCC. where i ranges over the entire dataset, including the releases with no smelly instances. Aggregate metrics are more robust than the mean, which is biased by the fact that datasets are unbalanced for different smell types in terms of smelly and non smelly instances (in some cases the datasets do not contain any smelly instance).

As a final step, we statistically verified the differences between the performance obtained by the experimented approaches. To this aim, we exploited the Wilcoxon test [324] computed on the distributions of MCC values of machine learning-based and heuristic-based techniques over the different releases and the different smell types. The results are intended as statistically significant at α=0.05. Furthermore, we estimated the magnitude of the measured differences by using Cliff's Delta (or d), a non-parametric effect size measure [56] for ordinal data. We followed well-established guidelines to interpret the effect size values: negligible for $|d| < 0.10$, small for $|d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [56].

3.2 ANALYSIS OF THE RESULTS

Table 3.5 shows the aggregate results for precision, recall, F-measure, and MCC achieved by the machine learning model ("NB" in the table) and DECOR. The overall results immediately highlight that the performance of both the approaches is generally low: indeed, the maximum F-measure is 41% and 44% for the NB and DECOR, respectively. This is especially due to the extremely low precision achieved over the entire dataset. More in detail, the high number of false positives is likely influenced by the fact that the dataset contains instances of different code smell types that sometimes have characteristics that may bias the detection approaches. As an example, let consider the cases of *God Class* and *Complex Class*. While the detection rules for these smells are different, it is reasonable to believe that some code metrics tend to have a similar distributions in the classes affected by those smells; for instance, being a *God Class* poorly cohesive and with a large number of methods, it is likely that also the complexity of the class tends to be high. This is the case of `taskdefs.optional.net.FTP` of the APACHE ANT 1.6.0 system, that is a *God Class* but has WMC=39. Such a value is not that high to make the class affected by *Complex Class* too, but it is enough to confound the machine learning technique, which wrongly signals the class as complex, thus giving a false positive. This result seems to suggest that an improved characterization of the symptoms behind specific code smell types

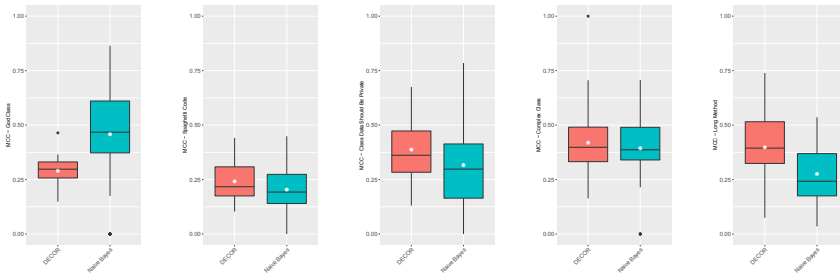

Figure 3.1: Boxplots representing the MCC values obtained by DECOR and NAIVE BAYESIAN (NB) for all the considered code smells

(e.g., by means of textual or historical analyses [170, 228, 234]) may make code smell detection more effective.

It is also worth to discuss the values achieved when considering the recall. In this case, we observed that DECOR is superior to NB in most cases: this confirms the experimental results obtained by the original authors [197] on the high recall of the approach. Finally, the low MCC values of both the approaches (see Figure 3.1) confirm that code metrics are not enough when it turns to code smell detection [38, 276].

Table 3.6: Comparison between NB and DECOR in terms of Wilcoxon and Cliff's Delta Effect Sizes. Significant p-values are reported in Bold Face.

	p-value	*d*	Meaning
God Class	<**0.01**	0.13	Negligible
Spaghetti Code	**0.01**	-0.29	Small
Class Data Should Be Private	<**0.01**	-0.43	Medium
Complex Class	**0.01**	-0.41	Medium
Long Method	<**0.01**	-0.33	Small

From a statistical point of view, Table 3.6 reports the results of the Wilcoxon and Cliff's delta tests computed on the MCC values of the experimented approaches. As shown, the difference between the techniques are statistically significant in all cases ($p-$values lower than 0.05). With the exception of *God Class*—where the machine learning model achieves higher MCC—all the other cases show that DECOR is statistically better than the baseline.

Table 3.7: Overlap between ML and DECOR in Absolute Terms and Percentages

	NB \ DECOR		NB ∩ DECOR		DECOR \ NB		NB ∪ DECOR		¬NB ∩ ¬DECOR	
	#	%	#	%	#	%	#	%	#	%
God Class	0	0	435	85	74	15	509	100	0	0
Spaghetti Code	14	1	419	29	266	18	699	48	744	52
Class Data Should Be Private	91	8	298	26	186	16	575	50	575	50
Complex Class	50	8	329	49	155	23	534	80	135	20
Long Method	1577	33	1076	23	650	14	3303	70	1460	30

Nevertheless, these differences are mostly negligible or have at most a medium effect.

In the following subsections, we discuss the findings achieved by considering each code smell independently and reporting qualitative examples aimed at further analyzing the performance of the detectors. Also, we discuss the complementarity between the sets of code smells correctly identified by the detectors (see 3.7), namely the extent to which NB and DECOR are able to identify the same instances.

3.2.1 *Results for God Class*

Looking at the results in both Table 3.5 and Figure 3.1, we can say that this smell is the easiest to detect and, in fact, all the instances affected by this smell have been correctly classified as smelly by at least one of the experimented techniques. In particular, we note that DECOR reaches a recall of 100%, which means that it is able to detect all *God Class* instances. Nevertheless, the high recall has a cost in terms of precision, that is just 8%. On the one hand, our findings are in line with those reported in previous studies [197, 228]. On the other hand, they confirm the need for further methodologies able to improve metric-based code smell detection.

When considering the complementarity of the approaches (Table 3.7), our results indicate a high overlap (85%): this means that in the vast majority of cases NB and DECOR can identify the same instances. However, a total of 74 actual *God Class* cases, corresponding to 15%, were only correctly identified by DECOR and missed by NB. To better understand the reasons that enable the heuristic approach to discover different instances than the machine learning

model, we went manually over the outputs of the techniques to conduct a manual analysis. As a result, we found that, while the heuristic approach can logically combine them obtaining better results, in most cases the machine learning model could be biased due to the characteristics of the training set, i.e., if the distributions of the involved independent variables are not representative in the training set a ML-based classifier could not be able to distinguish cases where certain conditions hold. As an example, let consider the class `CBZip2OutputStream` belonging to the project APACHE ANT 1.6.3: despite it is characterized by an LCOM < 20 (i.e., 19), it respects the rule reported in Table 3.3, as it has an ELOC = 2346 and NOM+NOA = 161. As such, the heuristic approach can still correctly identify its smelliness. On the contrary, the machine learning model classifies the instance as non-smelly.

3.2.2 *Results for Spaghetti Code*

As opposed to the previous case, this smell does not seem to be easily detectable automatically, as demonstrated by the results in Table 3.7, where we can see that more than half of the instances affected by this smell are not detected as smelly by any of the approaches. Overall, the performance achieved by the machine learning technique is extremely low, both in terms of precision and recall (15% and 30%, respectively). At the same time, DECOR has a higher recall (+17% with respect to other technique), but a lower precision (-4%), which had the effect to make the overall results of the two approaches comparable (F-Measure for DECOR is just 2% lower than NB, while MCC is 2% higher). Thus, we can claim that the metric-based detection is not able to provide good results, independently from the underlying technique adopted. Once again, this may indicate the need for further work aimed at improving the characterization of this smell type. Looking at the complementarity, also in this case the overlap is higher than the number of instances correctly detected by only one of the two. However, in the remaining cases DECOR is able to identify 266 code smell instances (18%) that are not correctly detected by the machine learning model. For example, the class `MeridioAuthority` of the APACHE INCUBATING 0.3

project is correctly detected only by DECOR. Basically, the reason is exactly the same observed before: the value of the ELOC metric of this class is 661, which is very close to the threshold (i.e., 600). While the heuristic technique discriminates based on thresholds, thus identifying the smelly class, the ML approach might be confounded by borderline metric values.

3.2.3 *Results for Class Data Should Be Private*

As for this smell, half of the smelly instances are not identified by any of the two techniques: this seems to be a clear indication of the need for more effective detection strategies. Between the two experimented techniques, DECOR is the one with the highest recall and this allows it to be slightly better than the machine learning model, overall. This is likely due to the very simple, yet clearer, detection rule applied by DECOR to identify instances of this smell. Conversely, the machine learning model seems to be less stable both in terms of precision and recall because it may be confounded by borderline values. The results shown in Table 3.7 confirm that the prediction model can only identify a limited number of instances that are not identified by DECOR (91), while in most cases there is an overlap (298) or the heuristic approach works better (186).

3.2.4 *Results for Complex Class*

The detection rule for this smell is only based on WMC, so the only factor that can determine different predictions is the value of this metric. First, we can confirm the results discussed so far, with DECOR having a high recall but a low precision. Moreover, the MCC of both the approaches is slightly in favor of DECOR (0.37 vs 0.36), indicating that (i) there is not a clear winner between the two and (ii) more sophisticated techniques for the detection of this smell would be worthwhile. The discussion of the overlap metrics is also very similar to the other smells discussed above. In general, we observed that the values that bring to an erroneous detection are the ones close to the threshold boundaries. The impact of boundary values can be also analyzed by

another point of view. Let consider the class `ServerSession` in the APACHE CASSANDRA 0.8.0 project, which shows a WMC = 47 that can be considered high but does not exceed the threshold of 50. In this case, the ML technique correctly detects the instance as smelly, while DECOR cannot.

3.2.5 *Results for Long Method*

As for *Long Method*, this was the only case in which the machine learning technique had higher recall than DECOR (i.e., +19%). Also for this code smell, the differences between the two approaches mainly concern instances with metric values close to the thresholds used by DECOR. An example is provided by the method `doSnapshot` belonging to the class `BlobStoreIndexShardGateway` of the *Elasticsearch* 0.17.0 project. The method under considerations has 79 lines of code and takes only 1 parameter. It is correctly detected by the prediction model as smelly but not by DECOR because it requires 100 or more lines of code and more than one parameter to identify the smell.

3.3 CONCLUSIONS

In this chapter we compared a machine learning approach with a heuristic one for code smell detection, considering five different types of code smells over a dataset composed of 125 releases of 13 open source software systems. Other than providing evidence that heuristic approaches perform better than machine learning ones, even if still with limited performance, our study highlighted some limitations, such as data imbalance or poor precision, that make the application of both machine learning models and heuristic approaches for code smell detection hard.

4

A LARGE EMPIRICAL ASSESSMENT OF THE ROLE OF DATA BALANCING IN MACHINE-LEARNING-BASED CODE SMELL DETECTION

This chapter presents an extensive empirical study in which we compare the performance of five data-balancing techniques for code smell detection with respect to a *no-balancing* baseline. To increase the generalisability of the results, we analyse two subsets of code smells extracted from two catalogues: (i) the catalogue proposed by Fowler *et al.* [88] for Object-Oriented code, and (ii) the catalogue proposed by Aniche *et al.* [10] for systems implementing the Model-View-Controller pattern. Our goal is understanding to what extent data balancing techniques can improve the accuracy of machine learning for code smell detection and which algorithms practitioners should use. This chapter provides the following contributions:

1. A large empirical study in which we exploit machine learning-based techniques to detect 11 code smell types for Object-Oriented systems on 125 releases of 13 software systems.

2. A second empirical study that includes four code smells to detect maintainability issues in Model-View-Controller systems [10]. Specifically, we analyse 120 projects relying on the Spring framework to answer two additional research questions.

3. A deep analysis on the role of balancing techniques and the impact of metrics selection.

4. A performance analysis in which we inspect the overhead in terms of efficiency caused by data balancing.

Table 4.1: Object-Oriented code smells along with the heuristics used to detect them and the features used by the ML models

Code Smell	Detection Rule	ML Model Features
God Class	$ELOC > \alpha \wedge (WMC + NOA) > \beta \wedge LCOM > \gamma$	$ELOC, WMC, NOA, LCOM$
Spaghetti Code	$ELOC > \alpha \wedge NMNOPARAM > \beta$	$ELOC, NMNOPARAM$
Class Data Should Be Private	$NOPA > \alpha$	$NOPA$
Complex Class	$McCabe > \alpha$	$McCabe$
Long Method	$LOC\_METHOD > \alpha \wedge NP \geq \beta$	LOC_METHOD, NP
Feature Envy	$MC > \alpha \wedge ATFD > \beta$	$MC, ATFD$
Inappropriate Intimacy	$(FanIn + FanOut) > \alpha$	$FanIn, FanOut$
Middle Man	$PDM > \alpha$	PDM
Refused Bequest	$PRM > \alpha$	PRM
Speculative Generality	$NOC > \alpha$	NOC
Long Parameter List	$NP > \alpha$	NP

4.1 DETECTION OF OBJECT-ORIENTED CODE SMELLS

The purpose of this study is to understand the impact of data balancing techniques on the accuracy of machine learning algorithms in detecting the design flaws from the catalogue designed by Fowler [88] who introduced the term code smell and adopted it for Object-Oriented systems. We aim to address the following research questions:

RQ$_1$. *Do data balancing techniques improve the effectiveness of machine learning-based detectors of code smell defined for Object-Oriented systems?*

RQ$_2$. *Which data balancing technique is the most effective at improving the effectiveness of machine learning-based code smell detectors for Object-Oriented systems?*

4.1.1 *Code Smells for Object-Oriented systems*

Code smells are "symptoms of poor design and implementation choices" [88] that have been widely observed to both analyse their characteristics [14, 47, 219, 252, 302, 304] and assess their impact on software maintainability [1,

Table 4.2: Complete list of the considered metrics for the detection of Object-Oriented code smells.

Acronym	Full Name	Smells
ATFD	Access To Foreign Data	Feature Envy
ELOC	Effective Lines Of Code	God Class, Spaghetti Code
FanIn	Max number of references to the subject class from another class in the system	Inappropriate Intimacy
FanOut	Max number of references from the subject class to another class in the system	Inappropriate Intimacy
LCOM	Lack of COhesion in Methods	God Class
LOC_METHOD	Lines Of Code of METHOD	Long Method
McCabe	McCabe's Cyclomatic Complexity	Complex Class
MC	Method Calls	Feature Envy
NOA	Number Of Attributes	God Class
NOC	Number Of Children	Speculative Generality
NOM	Number Of Methods	God Class
NOPA	Number Of Public Attributes	Class Data Should Be Private
NP	Number of Parameters	Long Method, Long Parameter List
NMNOPARAM	Number of Methods with NO PARAMeters	Spaghetti Code
PDM	Percentage of Delegated Methods	Middle Man
PRM	Percentage of Refused Methods	Refused Bequest
WMC	Weighted Methods Count	God Class, Complex Class

136, 224, 227, 288, 329, 331]. For many of these code smells heuristic detection rules have been defined [88, 191, 197, 226] based on metrics and thresholds to discriminate whether a component is smelly or not. We use the same metrics used by these heuristic detection rules to build machine learning models for code smell detection. In particular, we consider 11 code smells defined by Fowler [88] that are reported in Table 4.1 along with their detection rules, and lists of the metrics used in the Machine Learning models. The full description of such metrics is shown in Table 4.2.

4.1.2 *Data Balancing Techniques for Machine Learning*

The goal of the experiment is to compare the accuracy of different data balancing techniques, namely *Oversampling [168]*, *Undersampling*, *Synthetic Minority Oversampling TEchnique [48]*, *Cost-Sensitive Classifier [150]*, and *One-Class Classifier* [297]. Descriptions of the adopted data balancing techniques are reported in Section 2.2.

To this aim, we configure five different model variants based on the Naive Bayes classifier [126] which in our previous study showed to be the most effective in code smell detection. Our baseline consists of models trained without applying any data balancing technique (*No-balancing*). A dataset is imbalanced when its classes are not equally represented.

Table 4.3: Distribution statistics for Object-Oriented code smells

Code Smell	Min	Mean	Median	Max	Total
God Class	0	5.5	4	24	509
Spaghetti Code	0	12.7	11	31	1443
Class Data Should Be Private	0	11.4	11	37	1150
Complex Class	0	6.4	4	20	669
Long Method	3	48.3	26	147	4763
Feature Envy	0	1.3	0	12	148
Inappropriate Intimacy	0	15.4	2	774	1788
Middle Man	0	0.9	0	6	107
Refused Bequest	0	6.5	4	22	750
Speculative Generality	0	9.5	7	38	1106
Long Parameter List	0	5	1	29	578

4.1.3 *Subject Systems*

We select software systems for which a validated dataset of code smells exists. Specifically, we relied on 125 releases of 13 open-source software systems [226]. We employed the same dataset that we used in our previous study, presented in Chapter 3. The systems are heterogeneous since they have different sizes, lifetimes, and belong to different application domains. Note that the dataset consists of *manually validated* code smells instances (i.e., $8,534$). Looking at the distribution of code smells in the dataset, reported in Table 4.3, we can observe that each considered release have a low median number of code smells thus demonstrating once again that code smell detection is a highly imbalanced problem.

4.1.4 *Model Building and Evaluation*

For each model we apply a Feature Selection step by using CORRELATION-BASED FEATURE SELECTION (CFS) [113] to remove highly correlated independent variables. Then, we tune the hyper-parameters of the classifier by applying the GRID SEARCH algorithm [29], therefore resulting in five models that only differ for the choice of the data balancing technique to adopt.

As *independent variables* we consider the code metrics related to the structural characteristics of the software instances (e.g., size, complexity). We exploit the set of metrics originally adopted by Moha *et al.* [197]. In particular, given the smell detection rule, we design a model where we employ as independent variables only the metrics used in the detection rule. For example, for *God Class* the detection rule is $ELOC > 500 \wedge (WMC + NOA) > 20 \wedge LCOM > 350$. Therefore, we train the model on the effective number of lines of code (i.e., $ELOC$), the weighted methods per class (i.e., WMC), the number of attributes (i.e., NOA), and the lack of cohesion per class (i.e., $LCOM$). Table 4.1 reports the features used to detect each smell, while the complete list, including the full name of the metrics, is depicted in Table 4.2.

Since we are interested in detecting code smells, we set the presence/absence of a specific code smell as *dependent variable* of the machine learning model. This information was already available in the considered dataset.

To assess the capabilities of each of the five resulting machine learning models, we adopt 10-Fold Cross Validation [286]. The process is repeated 10 times, using each time a different fold as the test set. For each software system and data balancing technique, we build a machine learning model (i.e., within-project classification). The result consists of a confusion matrix for each code smell type, for each of the 125 project releases and each experimented classifier. Later, these matrices have been analysed to measure the evaluation metrics described in the following parts of the section.

To assess the effectiveness of the experimented detection techniques we compute four well-known metrics [16, 256], namely, *precision*, *recall*, *F-Measure*, and *Matthews Correlation Coefficient (MCC)*. We chose to discuss

results only in terms of MCC because this metric provides a better overview with respect to the other metrics by considering all the confusion matrix [271].

Since we consider several datasets, we need to aggregate the results achieved to have a more precise overview of the quality of results [12]. This step has been performed in a two-fold manner (i) by aggregating the confusion matrices and (ii) by plotting the results as boxplots. Boxplots are very useful to describe the distribution of the results and provide preliminary outcomes on the comparison of different techniques. However, to draw more reliable conclusions, they need to be complemented with statistical tests. Therefore, we use the Nemenyi test [214] for statistical significance and report its results by mean of MCB (Multiple comparisons with the best) plots [148]. We set the significance level to 0.05. The elements plotted above the gray band are statistically larger than the others.

4.1.5 *Results of the Study*

For each code smell, we first discuss the results by displaying boxplots, and then we evaluate their statistical significance relying on the results provided by the Nemenyi test. Note that we discarded all the cases in which at least one technique fails.

Figure 4.1 reports the boxplots for the MCC values obtained by applying different balancing techniques. The results of the Nemenyi test, for the statistical significance, are shown in Figure 4.2.

The first aspect we can observe is that, regardless of the balancing technique and the code smell under analysis, MCC values are between 0 and 0,5 which indicates that machine learning has limited accuracy for Object-Oriented code smell detection.

The results show that *SMOTE* is the most effective technique. However, in 7 out of 11 cases, none of the balancing techniques is significantly better in terms of MCC. *No-balancing* provides good accuracy as well, since it appears six times in the group containing the most effective techniques.

An important aspect to remark is that for 4 out of 11 object-oriented code smells, *SMOTE* and *No-balancing* MCCs are significantly higher than all the

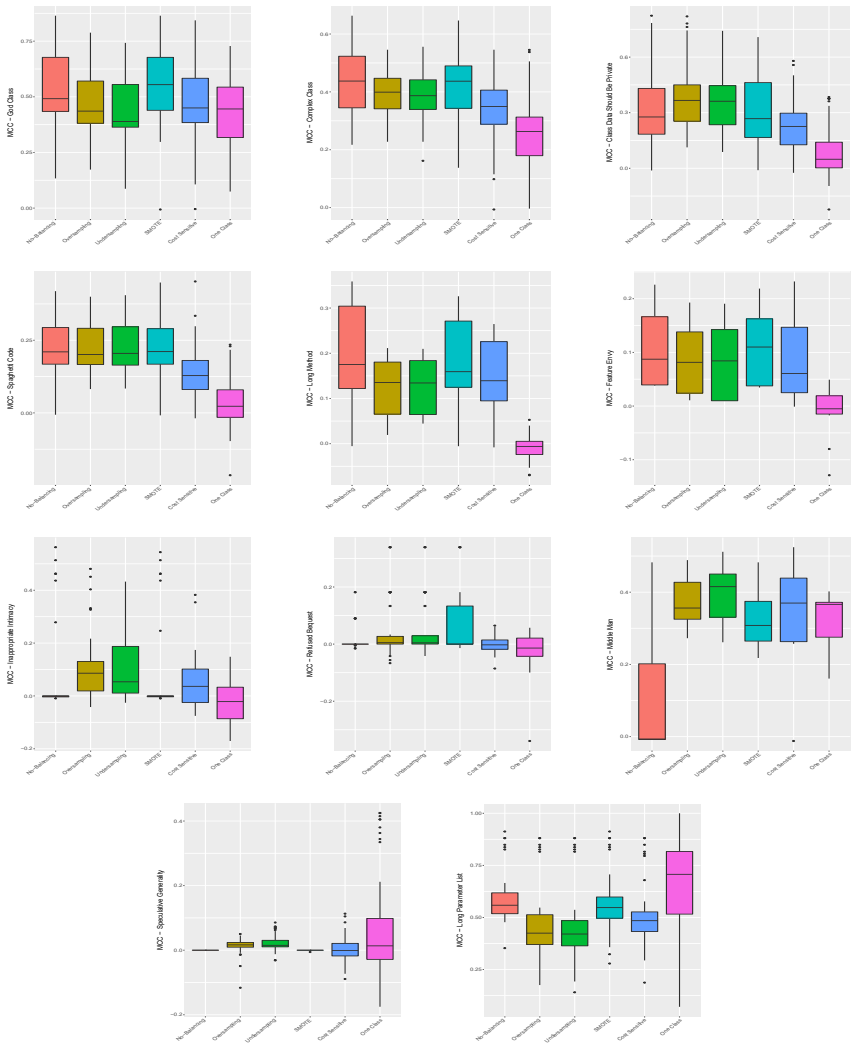

Figure 4.1: Boxplots representing the MCC values obtained by NAIVE BAYESIAN trained applying different balancing strategies for Object-Oriented code smells detection.

other balancing techniques. This is the case of two class-level code smells (*God Class*, and *Complex Class*) and two method-level code smells (*Long Method*, and *Feature Envy*). *God Class* and *Complex Class* are the easiest class-level code smells to identify. Their detection rules are straightforward

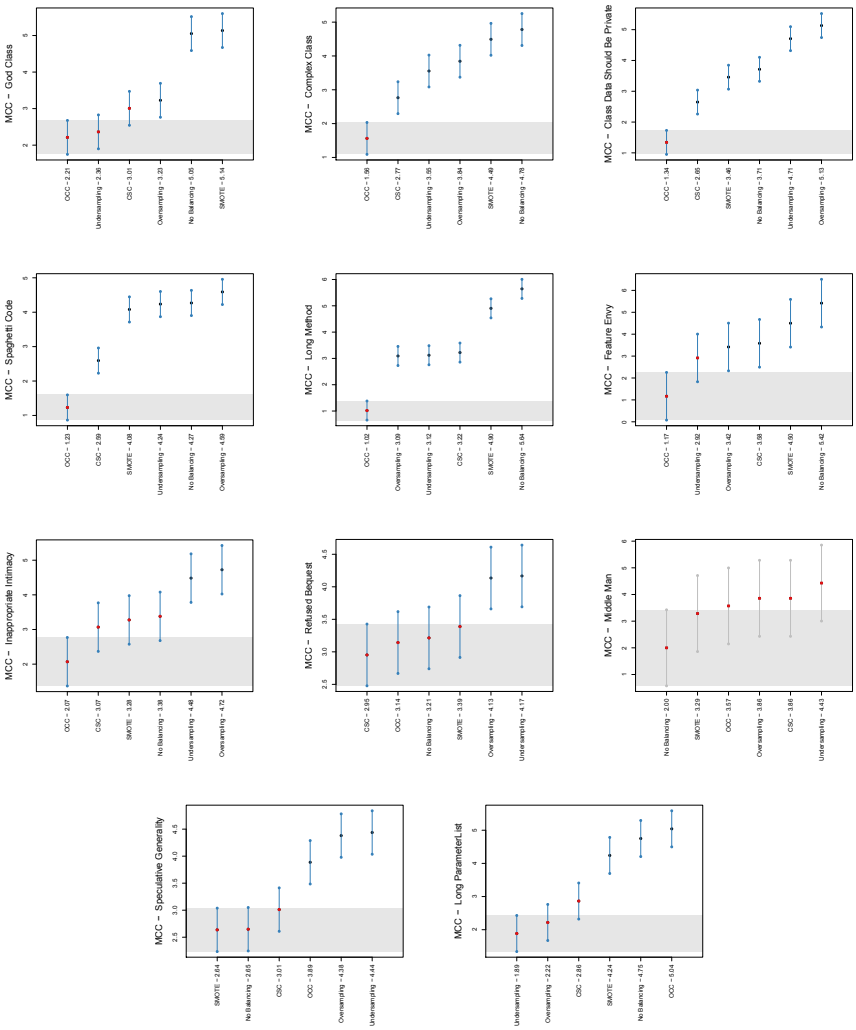

Figure 4.2: Plots representing the results of Nemenyi test for statistical significance between the MCC values obtained by NAIVE BAYESIAN trained applying different balancing strategies for Object-Oriented code smells detection.

and based on easy-to-calculate metrics (e.g., size, complexity), leading to a median MCC close to 0.5 regardless of the data balancing applied. As for *Long Method* and *Feature Envy*, these are method-level smells; hence, the total number of instances to predict is much higher. We could deem

SMOTE and *No-balancing* to have higher effectiveness where the detection metrics are simple or a large number of instances is available. However, *LongParameterList* is an exception. Indeed, although it is a method-level smell, the best MCC values are achieved by *One-Class Classifier*. In this specific case, *SMOTE* and *No-balancing* accuracy is slightly lower than *One-Class Classifier* but still better than all the other techniques.

Two unusual cases for which a specific discussion is required are *Middle Man* and *Speculative Generality*. *Middle Man* represents one of the rare cases in which data balancing techniques improve ML effectiveness. Indeed, *No Balancing* is the least accurate technique, with a quite clear difference to the others. As for *Speculative Generality*, results show that, regardless of the adopted data balancing technique, MCC values are very low proving that machine learning is still not applicable for the detection of this smell with the set of metrics used in our study.

By and large, results suggest that there is no balancing technique which is better than the others. Indeed, different balancing techniques could be more suitable for different types of code smells. However, the highest accuracy is achieved by *No-balancing* and *SMOTE*, except for some code smells in which *One-Class Classifier* shows a higher MCC.

Summing Up: The results show that the performance of current machine learning-based approaches for detecting Object-Oriented code smells is quite limited, regardless of the adopted balancing technique ($MCC < 0.50$). Overall, *SMOTE* and *No Balancing* seem to be more effective than the other techniques.

4.2 DETECTION OF MODEL-VIEW-CONTROL CODE SMELLS

The purpose of the second study is to understand the impact of data balancing techniques on the accuracy of machine learning algorithms in detecting the design flaws from the catalogue designed by Aniche *et al.* [10] who defined smells specific for systems implementing the Model-View-Control pattern.

Table 4.4: MVC code smells along with the heuristics used to detect them and the features used by the ML models

Code Smell	Detection Rule	ML Model Features
Brain Repository	$McCabe > \alpha \wedge SQLC > \beta$	$McCabe, SQLC$
Fat Repository	$CTE > \alpha$	CTE
Promiscuous Controller	$NSR > \alpha \vee NSD > \beta$	NSR, NSD
Brain Controller	$NFRFC > \alpha$	$NFRFC$

Table 4.5: Complete list of the considered metrics for the detection of Model-View-Controller code smells.

Acronym	Full Name	Smells
McCabe	McCabe's Cyclomatic Complexity	Brain Repository
NOR	Number of Routes	Promiscuous Controller
NSD	Number of Services as Dependencies	Promiscuous Controller
NFRFC	Non-Framework Response For a Class	Brain Controller
SQLC	SQL Complexity	Brain Repository
CTE	Calls to Entities	Fat Repository

Specifically, we aim at addressing the same research questions as for the Object-Oriented code smells:

RQ$_{3.}$ *Do data balancing techniques improve the effectiveness of machine learning algorithms in detecting code smells specific for systems implementing the Model-View-Controller pattern?*

RQ$_{4.}$ *Which data balancing technique is the most effective at improving the effectiveness of machine learning algorithms in detecting code smells specific for systems implementing the Model-View-Controller pattern?*

4.2.1 *Code Smells*

We analyse the code smells specific to systems adopting the Model-View-Controller pattern [10]. Such a pattern is popular across many well-know

frameworks (e.g., RUBY ON RAILS, SPRING MVC, ASP.NET MVC) [10]. In particular, we consider four code smells for which we report the heuristics needed to detect them and the metrics that we used to build the machine learning models in Table 4.4. Such metrics are fully described in Table 4.5.

4.2.2 *Data Balancing Techniques for machine learning*

We experiment the same base classifier (i.e., Naive Bayes) and the same set of data balancing techniques previously used in Section 4.1.

Table 4.6: Distribution statistics for MVC code smells

Code Smell	Min	Mean	Median	Max	Total
Brain Repository	0	0.5	0	26	31
Fat Repository	0	1.2	0	28	126
Promiscuous Controller	0	6.7	0	478	682
Brain Controller	0	1.1	0	14	66

4.2.3 *Subject Systems*

We use the dataset developed by Aniche *et al.* [10], consisting of 120 open-source systems. We rely on this dataset because the approach used to detect the smells has been validated with expert industrial developers in software systems implemented using Spring. This widely adopted MVC framework uses stereotypes to explicitly mark classes playing different roles (e.g., Controller classes), thus facilitating identifying the role of each class. The distribution of the smells is reported in Table 4.6.

4.2.4 *Model Building and Evaluation*

We build and evaluate the models by following the same procedure described in Section 4.1.4 except the *independent variables* that were extracted from the heuristics derived by Aniche *et al.* [10]. Table 4.4 reports the features

used to detect each smell, while the complete list, including the full name of the metrics, is depicted in Table 4.5.

As for Object-Oriented code smells, we first discuss the results by analysing the boxplots and then verify their statistical significance relying on the Nemenyi test [214]. Please consider that, also in this case, we discarded all the cases in which at least one technique fails.

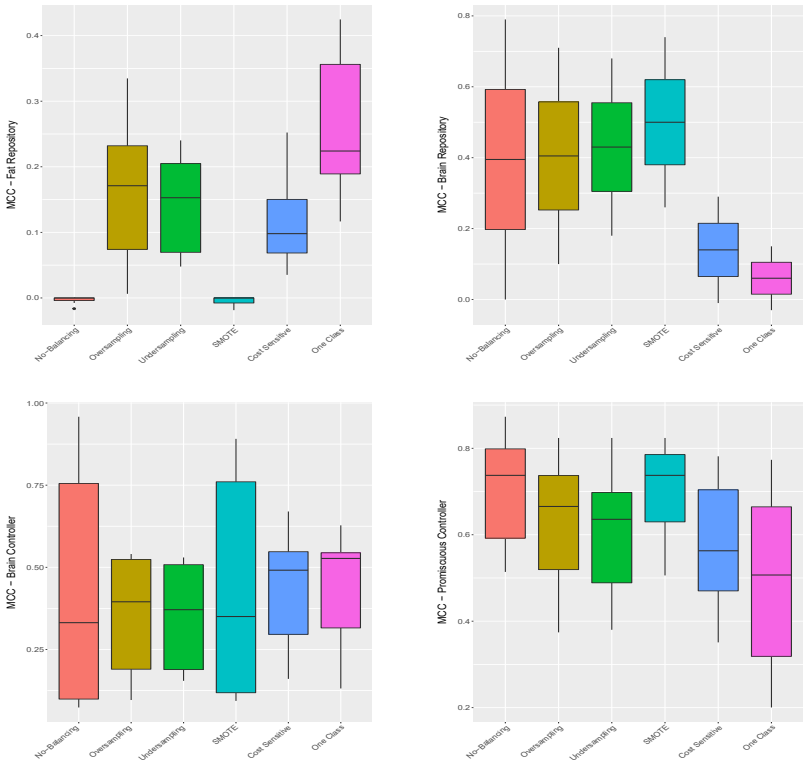

Figure 4.3: Boxplots representing the MCC values obtained by NAIVE BAYESIAN trained applying different balancing strategies for MVC code smells detection.

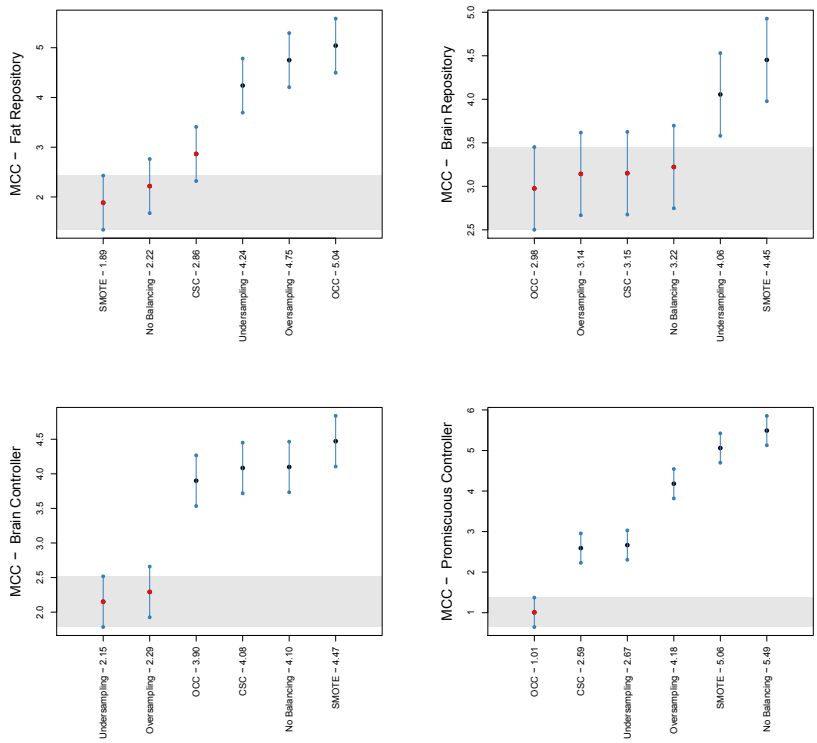

Figure 4.4: Plots representing the results of Nemenyi test for statistical significance between the MCC values obtained by NAIVE BAYESIAN trained applying different balancing strategies for MVC code smells detection.

4.2.5 *Results of the Study*

The results for MVC code smell detection reported in Figure 4.3 and Figure 4.4 show that ML has pretty higher accuracy when detecting this type of code smells than when detecting Object-Oriented code smells (i.e., MCC values up to ≈ 0.70). Similarly to the Object-Oriented case, *SMOTE* and *No-balancing* show higher accuracy with respect to the other balancing techniques. As already observed, these two balancing techniques seem to be more effective where the ML algorithm has a higher prediction power. Indeed *SMOTE* achieves significantly higher MCCs in all cases except for *Fat Repository*

in which MCC values are lower. Instead, *No-balancing* appears in the first group in 2 out of 4 cases. A singular case is the *Fat Repository* smell, where *One-Class Classifier* accuracy is significantly higher than the other balancing techniques. A possible motivation behind this surprising result could be found by analysing the smell distribution in Table 4.6. Indeed, the class distribution for *Fat Repository* is almost uniform (i.e., the smelly instances are well spread across the project). Therefore, in most of the cases there are enough instances to build a reliable training set.

A final consideration is that MVC code smells are likely to be more system-dependent. Boxplots indicate a high variability of results with respect to the considered system showing very large distributions in most of the cases.

Summing Up: With respect to the Object-Oriented case, machine learning-based approaches are sharply more effective for the detection of MVC related code smells. In three out of four cases, the results are pretty good, achieving MCC values up to 0.67 and recall up to 1.00. Similarly to Object-Oriented systems, *No Balancing* and *SMOTE* are the most effective techniques.

4.3 CONCLUSION

In this chapter, we have presented a large-scale empirical comparison between six different balancing techniques for Machine-Learning-based code smell detection. The study considered eleven code smells for Object-Oriented systems and four code smells for systems implementing the Model-View-Controller pattern. For the former, we relied on a manually-validated dataset comprising 125 releases belonging to 13 open source systems. In contrast, for the latter, our dataset consisted of 120 Spring Model-View-Controller Open Source Systems.

The results suggest that Machine-Learning models relying on *SMOTE* achieve the best accuracy. However, its training phase is not always feasible in practice. Furthermore, avoiding balancing does not dramatically impact effectiveness. Techniques which perform training only on the minority class

(i.e., *Cost-Sensitive Classifier* and *One Class Classifier*), and resampling techniques (i.e., *Class Balancer* and *Resample*) are both not effective. Existing data balancing techniques are therefore, inadequate for code smell detection. Furthermore, the results indicate that structural metrics alone are not adequate for code smell detection, confirming the previous work [228, 234] on the necessity of textual and historical metrics as well as their combination with structural metrics to achieve better accuracy. This hinders the feasibility of the current Machine-Learning-based approaches.

5

ON THE ADEQUACY OF STATIC ANALYSIS WARNINGS WITH RESPECT TO CODE SMELL DETECTION

In this chapter, we conduct a preliminary, motivational investigation into the actual relation between static analysis warnings and code smells, also attempting to assess the potential predictive power of those warnings.

Then, we analyze the performance of code smell detection techniques based machine learners and using the static analysis warnings as features. Then, we further investigate the problem by studying the overlap among the predictions made by machine learning models built using the warnings of different static analysis tools as features: such an analysis reveals a high complementarity suggesting that a combination of those warnings could potentially improve the code smell detection capabilities. As such, we define and experiment a new combined model which significantly perform better than the individual models.

In the last part of our study, we go beyond and analyze how this combined model can be further combined with additional code metrics that have been used for code smell detection in previous work [15]. While the performance of the combined model significantly performs better than previous approaches based on software metrics.

To sum up, this chapter provides the following contributions:

1. A preliminary analysis on the suitability of static analysis warnings in the context of code smell detection;

2. An empirical understanding of how machine learning techniques for code smell detection work when fed with warnings generated by automated static analysis tools;

3. A machine learning-based detector that combines multiple automated static analysis tools, improving on the performance of individual detectors;

4. An empirical understanding of how warning-based machine learning techniques for code smell detection work in comparison with metric-based ones;

5. A machine learning-based detector that combines static analysis warnings and code metrics, further improving detectors' performance;

5.1 RESEARCH METHODOLOGY

In the context of this empirical study, we had the ultimate goal of assessing the extent to which static analysis warnings can contribute to the identification of design issues in source code. We faced this goal by means of multiple analyses and research angles.

We defined three main dimensions. At first, we conducted a statistical study aiming at investigating whether and to what extent can static analysis warnings be actually used and useful in the context of code smell detection. Such an analysis must be deemed as preliminary, since it allowed us to quantify the potential benefits provided by those warnings: should this have not provided sufficiently acceptable results, this would have already stopped our investigation. On the contrary, a positive result would have provided further motivations into the need for a closer investigation on the role of static analysis warnings for code smell detection.

In this regard, we defined the first two research questions. In the first place, we aimed at assessing if the distribution of static analysis warnings differs when computed on classes affected and not affected by code smells. Rather than approaching the problem from a correlation perspective, we preferred to use a distribution analysis since the latter may provide insights on the specific types of warnings that are statistically different in the two sets of classes, i.e., smelly or smelly-free—on the contrary, correlations might have only given an

indication of the strength of association, without reporting on the statistical significance when computed on smelly and non-smelly classes. We asked:

RQ$_1$. *How do static analysis warning types differ in classes affected and not affected by code smells?*

In the second place, we complemented the distribution analysis with an additional investigation into the potential usefulness of static analysis warnings for code smell detection. While the first preliminary analysis had the goal to assess the distribution of warnings in classes affected or not by code smells, this second step aimed at quantifying the contribution that such warnings might provide to code smell detection models. In particular, we asked:

RQ$_2$. *How do static analysis warnings contribute to the classification of code smells?*

Once we had ensured the feasibility of a deeper analysis, we then proceeded with the investigation of the performance achieved by a code smell detection model relying on static analysis warnings as predictors. This analysis allowed us to provide quantitative insights on the actual usefulness of static analysis warnings, other than understanding their limitations when considered in the context of code smell detection. This led to the definition of three additional research questions.

First, on the basis of the results achieved in the preliminary study, we devised machine learning-based techniques—one for each static analysis tool considered, as explained later in this section—that exploit the warnings providing more contribution to the classification of code smells. Afterwards, we assessed their performance by addressing **RQ$_3$**:

RQ$_3$. *How do machine learning techniques that exploit the warnings of single static analysis tools perform in the context of code smell detection*

Once we had assessed the classification performance of the individual models created in **RQ$_3$**, we discovered that these models had low performance, especially due to false positives. To overcome this issue, we moved toward the analysis of the complementarity between the individual models, namely the extent to which different models could identify different code smell instances.

This was relevant because a positive answer could have paved the way to a combination of multiple models. Hence, we asked:

RQ$_4$. *What is the orthogonality among the individual machine learning-based code smell detectors?*

Given the results achieved when addressing **RQ$_4$**, we then devised a combined model. The process required the identification of the optimal subset of the static analysis warnings exploited by different tools. While investigating the performance of such a combined model, we addressed **RQ$_5$**:

RQ$_5$. *How do machine learning techniques that combine the warnings of different static analysis tools perform in the context of code smell detection?*

The analyses defined so far could help understand how static analysis warnings enable the identification of code smells. Yet, it is important to remark that the research on machine learning for code smell detection has been vibrant over the last years [15] and, as a matter of fact, a number of researchers has been working on the optimization of machine learning pipelines with the goal of improving the code smell detection capabilities. We took into account this aspect when defining the third part of our investigation. The last part of the empirical study consisted of the definition of the last three research questions.

First, we compared the best machine learner coming from the previous study, namely the one that combines the static analysis warnings coming from different tools, with a machine learner that exploits structural code metrics, namely a state of the art solution that has been used multiple times in the past [15]. This led to the formulation of our **RQ$_6$**:

RQ$_6$. *How does the combined machine learner work when compared to an existing, code metrics-based approach for code smell detection?*

Afterwards, we proceeded with a complementarity analysis involving the two techniques (i.e., the combined machine learner and the metrics-based approach for code smell detection) in order to understand to what extent the models built on two different sets of metrics could identify identify different code smell instances. In case of a positive answer, better performance could

be achieved by combining these two sets of metrics together. In this regard, we asked the following research question:

RQ$_7$. *What is the orthogonality among the combined machine learner and the metrics-based approach for code smell detection?*

Finally, after we have studied the complementarity between the two models, we evaluated an additional combination, which aimed at putting together static analysis warnings and code metrics. Hence, we asked:

RQ$_8$. *How do machine learning techniques that combine static analysis warnings and code metrics perform in the context of code smell detection?*

The next sections report on the data selection, collection, and analysis procedures adopted to address our research questions.

5.1.1 *Context of the Study*

The *context* of the study was composed of open-source software projects, code smells, and static analysis tools.

5.1.1.1 *Selection of Code Smells*

The exploited dataset reports code smell instances pertaining to 13 different types. However, not all of them are suitable for a machine learning solution. For instance, let consider the case of *Class Data Should Be Private*: this smell appears when a class exposes its attributes, i.e., the attributes have a `public` visibility. By definition, instances of this code smell can be effectively detected using simpler rule-based mechanisms, as done in the past [197].

For this reason, we first filtered out the code smell types whose definitions do not require any threshold. In addition, we filtered out method-level code smells, e.g., *Long Method*. The decision was driven by three main observations. In the first place, the vast majority of the previous papers on code smell detection have used a class-level granularity [15] and, therefore, our choice allowed for a simpler interpretation and comparison of the results. Secondly, our study focuses on the code smells perceived by developers

as the most harmful [227, 288], which are all at class-level. Thirdly, the analyses performed in the context of our empirical study required the use of a heuristic code smell detector (i.e., DECOR [197]) that has been designed and experimentally tested on class-level code smells. All these reasons led us to conclude that considering method-level code smells would not be necessarily beneficial for the study. Nonetheless, our future research on the matter will consider the problem of assessing the role of static analysis warnings for the detection of method-level code smells.

Based on these considerations, we focused our study on seven code smells, namely **God Class**, **Spaghetti Code**, **Complex Class**, **Inappropriate Intimacy**, **Lazy Class**, **Refused Bequest**, and **Middle Man**. Detailed descriptions of code smells are reported in Table 2.1

The selected code smells are those more often targeted by related research [15]. They have been also connected to an increase of change- and fault-proneness of source code [42, 137, 226] as well as maintenance effort [277]. According to previous work [137, 226, 329], all the code smells considered let the affected source code be more prone to changes and faults in different manners. As an example, Palomba et al. [226] reported that the change-proneness of classes affected by the *God Class* smell is around 28% higher than classes not affected by the smell, while *Spaghetti Code* increases the change-proneness of classes of about 21%. Other empirical investigations provided different indications, e.g., Khomh et al. [135, 137] reported that 68% of the classes affected by a *God Class* are also change-prone. As a matter of fact, our current body of knowledge reports that all the code smells we considered are connected to change- and fault-proneness, but different studies provided different estimations on the extent of such connection. In addition, these code smells are highly relevant for developers that, indeed, often recognize them as harmful for the evolvability of software projects [227, 288, 330].

5.1.1.2 *Selection of Automated Static Analysis Tools*

In the context of our research, we selected three well-known automated static analysis tools such as CHECKSTYLE, FINDBUGS, and PMD. We provide a brief description of these tools in the following:

- **Checkstyle**. CHECKSTYLE is an open-source developer tool that evaluates JAVA code according to a certain coding standard, which is configured according to a set of "checks". These checks are classified under 14 different categories, are configured according to the coding standard preference, and are grouped under two severity levels: error and warning. More information regarding the standard checks can be found from the Checkstyle web site.[1]

- **Findbugs**. FINDBUGS is another commonly used static analysis tool for evaluating JAVA code, more precisely Java bytecode. The analysis is based on detecting "bug patterns", which arise for various reasons. Such bugs are classified under 9 different categories, and the severity of the issue is ranked from 1-20. Rank 1-4 is the *scariest* group, rank 5-9 is the *scary* group, rank 10-14 is the *troubling* group, and rank 15-20 is the *concern* group.[2]

- **PMD**. PMD is an open-source tool that provides different standard rule sets for major languages, which can be customized by the users, if necessary. PMD categorizes the rules according to five priority levels (from P1 "Change absolutely required" to P5 "Change highly optional"). Rule priority guidelines for default and custom-made rules can be found in the PMD project documentation.[3]

The selection of these tools was driven by recent findings reporting that these are among the automated static analysis tools more employed in practice by developers [159, 312, 315]. In particular, the most recent of these papers

1 https://checkstyle.sourceforge.io
2 http://findbugs.sourceforge.net/findbugs2.html
3 https://pmd.github.io/latest/

[315] reported that CHECKSTYLE, PMD, and FINDBUGS are actually the tools that practitioners use more when developing in JAVA, along with SONARQUBE. The selection was therefore based on these observations. In this respect, it is also worth remarking that we originally included SONARQUBE as well. However, we had to exclude it because it failed on all the projects considered in our study (see Section 5.1.1.3).

5.1.1.3 *Selection of Software Projects*

To address the research goals and assess the capabilities of the machine learning techniques for code smell detection, we needed to rely on a dataset reporting actual code smell instances. Most previous studies [15] focused on datasets collected using automated mechanisms, e.g., executing multiple detectors at the same time to consider the instances detected by all of them as actual code smells. Nonetheless, it has been shown that the performance of machine learning-based code smell detectors might be biased by the approximations done, other than by the false positive instances detected when building the ground truth of code smells [71]. In this study, we took advantage of these latter findings and preferred to rely on a manually-labeled dataset containing actual code smell instances. Of course, this choice might have had an impact on the size of the empirical study since there exist only a few datasets of manually-labeled code smells [15]. Yet, we were still convinced to opt for this solution, as this was the most appropriate choice to do in order to have reliable results. Indeed, a dataset of real smell instances allowed us to provide reliable results on the performance capabilities of the experimented models and, at the same time, to present a representative case of a real scenario where the code smells arise in similar amounts as in our study [226].

From a technical viewpoint, the selection of projects was driven by the above requirement. We exploited a publicly available dataset of code smells developed in previous research [226, 232]: this provides a list of 17,350 manually-verified instances of 13 code smell types pertaining to 395 releases of 30 open source systems. Given this initial dataset, we fixed two constraints that the projects to consider had to satisfy. First, the projects had to contain data for the code smells selected in our investigation (see Section 5.1.1.1).

Secondly, we required them to be successfully built so that they could be later analyzed by the selected static analysis tools (see Section 5.1.1.2). These two constraints were satisfied in 25 releases of the 5 open-source projects reported in Table 5.1 along their main characteristics.

Table 5.1: Software systems considered in the project.

Project	Description	# Classes	# Methods
Apache Ant	Build system	1,218	11,919
Apache Cassandra	Database Management System	727	7,901
Eclipse JDT	Integrated Development Environment	5,736	51,008
HSQLDB	HyperSQL Database Engine	601	11,016
Apache Xerces	XML Parser	542	6,126

For the sake of completeness, it is worth reporting that most of the excluded releases/projects were due to build issues, e.g., dependency resolution problems [303]. This possibly remarks the need for additional public code smell datasets composed of projects that can be analyzed through static or dynamic tools.

5.1.2 *Data Collection*

The data collection phase aimed at gathering information related to dependent and independent variables of our study. These concern the labeling of code smell instances, namely the identification of real code smells affecting the considered systems, and the collection of static analysis warnings from the selected analyzer, which will represent the features to be used in the machine learners designed in the empirical study.

5.1.2.1 *Collecting information on actual code smell instances*

This stage consisted of identifying real code smells in the considered software projects. The data collection, in this case, was inherited by the dataset exploited. While some previous studies relied on automated mechanisms for

Table 5.2: Descriptive statistics about the number of code smell instances.

Code Smell	Min.	Median	Mean	Max.	Tot.
God Class	0.00	4.00	6.19	23.00	412
Complex Class	0.00	2.00	4.27	16.00	301
Spaghetti Code	0.00	11.00	12.40	32.00	773
Inappropriate Intimacy	0.00	2.00	3.03	10.00	206
Lazy Class	0.00	1.00	1.95	11.00	141
Middle Man	0.00	1.00	1.11	6.00	84
Refused Bequest	0.00	7.00	7.35	17.00	500

this step, e.g., by using metric-based detectors [85, 135, 177], recent findings showed that such a procedure could threaten the reliability of the dependent variable and, as a consequence, of the entire machine learning model [71]. Hence, in our study we preferred a different solution, namely considering manually-validated code smell instances. For all the systems considered, the publicly available dataset exploited in the empirical study report actual code smell instances [226, 232] and has been used in recent studies evaluating the performance of machine learning models for code smell detection [226]. For each code smell, Table 5.2 reports the distribution of the code smells in the dataset.

5.1.2.2 *Collecting static analysis tool warnings*

This step aimed at collecting the data of the independent variables used in our study. Each tool required a different process to collect such data:

- **Checkstyle**. The `jar` file for the CHECKSTYLE analysis was downloaded directly from the Checkstyle's website[4] in order to engage the analysis from the command line. The version of the executable `jar` file used was the `checkstyle-8.30-all.jar`. In addition to downloading the `jar` executable, CHECKSTYLE offers two different types of rule sets for the analysis. For each of the rule sets, the configuration file was

4 https://checkstyle.org/#Download

downloaded directly from Checkstyle's guidelines.[5] In order to start the analysis, the `checkstyle-8.30-all.jar` and the configuration file in question were saved in the directory where all the projects resided.

- **Findbugs**. FINDBUGS 3.0.1 was installed by running the `brew install findbugs` in the command line. Once installed, the GUI was then engaged by writing `spotbugs`. From the GUI, the analysis was executed through *File* → *New Project*. The classpath for the analysis was identified to be the location of the project directory. Moreover, the source directories were identified to be the project `jar` executable. Once the class path and source directories were identified, the analysis was engaged by clicking Analyze in the GUI. Once the analysis finished, the results were saved through *File* → *Save as* using the XML file format. The main specifications were the "Classpath for analysis (jar, ear, war, zip, or directory)" and "Source directories (optional; used when browsing found bugs)" where the project directory and project `jar` file were added.

- **PMD**. PMD 6.23.0 was downloaded from GitHub[6] as a zip file. After unzipping, the analysis was engaged by identifying several parameters: project directory, export file format, rule set, and export file name. In addition to downloading the zip file, PMD offers 32 different types of rule sets for Java.[7] All 32 rule sets were used during the configuration of the analysis.

Using these procedures, we ran the three static analysis tools on the considered software systems. At the end of the analysis, these tools extracted a total of 60,904, 4,707, and 179,020 warnings for CHECKSTYLE, FINDBUGS, and PMD, respectively.

5 https://github.com/checkstyle/checkstyle/tree/master/src/main/resources
6 https://github.com/pmd/pmd/releases/download/pmd_releases%2F6.23.0/pmd-bin-6.23.0.zip
7 https://github.com/pmd/pmd/tree/master/pmd-java/src/main/resources/rulesets/java

5.1.3 *Data analysis*

In this section, we report the methodological steps conducted to address our research questions.

5.1.3.1 *RQ_1. Distribution analysis.*

To address the first research question, we first showed boxplots depicting the distribution of the metrics and smells. Then, we computed the Mann-Whitney and Cliff's Delta tests to verify the statistical significance of the observed differences and their effect size. With respect to other possible analyses methods (e.g., correlation), studying the distribution of warnings into the smelly and non-smelly classes not only allowed us to identify the warning types that are more related to code smells, but also to quantify the extent of the difference between the number of warnings contained in smelly and non-smelly classes.

5.1.3.2 *RQ_2 Contribution of static analysis warnings in code smell detection.*

In this **RQ**, we assessed the extent to which the various warning categories of the considered static analysis tools can potentially impact the performance of a machine learning-based code smell detector. To this aim, we employed an information gain measure [257], and particularly the *Gain Ratio Feature Evaluation* technique. This analysis method turned to be particularly useful in our case, since it allowed us to precisely quantify the potential predictive power of each warning category for the detection of code smells.

5.1.3.3 *RQ_3. The role of static analysis warnings in code smell detection.*

Once we had investigated which warning categories relate the most to the presence of code smells, in $\mathbf{RQ}_3$ we proceeded with the definition of machine learning models. Specifically, we defined a feature for each warning type raised by the tools, where each feature contained the number of violations of that type identified in a class. For instance, suppose that for a class C_i

CHECKSTYLE identifies seven violations to the warning type called *"Bad Practices"*: the machine learner is fed with the integer value "7" for the feature *"Bad Practices"* computed on the class C_i.

The dependent variable was, instead, given by the presence/absence of a certain code smell. This implied the construction of seven models for each tool, i.e., for each static analysis tool considered, we built a model that used its warnings types as features to predict the presence of *God Class*, *Spaghetti Code*, *Complex Class*, *Inappropriate Intimacy*, *Lazy Class*, *Refused Bequest*, and *Middle Man*. Overall, this design led to the creation of 21 models per project, i.e., one for each code smell/static analysis tool pair. For the sake of clarity, it is worth remarking that we considered each release of the projects in the dataset as an independent project. This choice was taken after an in-depth investigation of the differences among the releases available: we indeed discovered that the releases that met our filtering criteria (see Section 5.1.1.3) were too far in time from each other, making other strategies unfeasible/unreliable—as an example, the excessive distance among releases made not feasible a release-by-release methodology where subsequent releases are considered following a time-sensitive data analysis [248, 293].

As for the supervised learning algorithm, the literature in the field still misses a comprehensive analysis of which algorithm works better in the context of code smell detection [15]. For this reason, we experimented with multiple classifiers such as *J48*, *Random Forest*, *Naive Bayes*, *Support Vector Machine*, and *JRip*. When training these algorithms, we followed the recommendations provided by previous research [15, 293] to define a pipeline dealing with some common issues in machine learning modeling. In particular, we exploited the output of the Gain Information algorithm—used in the context of $\mathbf{RQ}_2$—to discard irrelevant features that could bias the interpretation of the models [293]: we did that by excluding the features not providing any information gain. We also configured the hyper-parameters of the considered machine learners using the MULTISEARCH algorithm [334]Finally, we considered the problem of data balancing: since our previous findings showed that data balancing may or may not be useful to improve the performance of a model, before deciding on whether to apply data balancing,

we benchmarked (i) *Class Balancer*, which is an oversampling approach (ii) *Resample*, an undersampling method (iii) *Smote*, an approach including synthetic instances to oversample the minority class, and (iv) *NoBalance*, namely the application of no balancing methods.

After training the models, we proceeded with the evaluation of their performance. We applied a 10-fold cross-validation, as it allows to verify multiple times the performance of a machine learning model built using various training data against unseen data. For each test fold, we evaluated the models by computing a number of performance metrics, such as precision, recall, F-Measure, and Matthews Correlation Coefficient (MCC). Finally, with the aim of drawing statistically significant conclusions, we applied the post-hoc Nemenyi test [214] on the distributions of MCC values achieved by the experimented machine learners, setting the significance level to 0.05.

5.1.3.4 *RQ$_4$. Orthogonality between the three single-tool Detection Models.*

When addressing this research question, we were interested in understanding whether the different machine learners experimented in the context of **RQ$_3$** were able to correctly identify the smelliness of different classes. If this was the case, then it meant that different automated static analysis tools would have had the potential to predict the smelliness of classes differently, hence possibly enabling the definition of a combined machine learning mechanism that it could have further improved the code smell detection capabilities. In other terms, the analysis aimed at understanding how many true positives can be identified by a specific model alone and how many true positives can be correctly identified by multiple models. To this purpose, for each code smell type, we compared the sets of *correctly* detected instances by a technique m_i with those identified by an alternative technique m_j using the following overlap metrics [220]:

$$correct_{m_i \cap m_j} = \frac{|correct_{m_i} \cap correct_{m_j}|}{|correct_{m_i} \cup correct_{m_j}|}\%$$

$$correct_{m_i \setminus m_j} = \frac{|correct_{m_i} \setminus correct_{m_j}|}{|correct_{m_i} \cup correct_{m_j}|}\%$$

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 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.

We also considered an additional orthogonality metric, which computed the percentage of code smell instances correctly identified only by the detection model m_i. In this way, we could measure the extent to which the warning types of a specific static analysis tool contributed to the identification of all correct instances identified. Specifically, we computed:

$$correct_{m_i \setminus (m_j \cup m_k)} = \frac{|correct_{m_i} \setminus (correct_{m_j} \cup correct_{m_k})|}{|correct_{m_i} \cup correct_{m_j} \cup correct_{m_k}|}\%$$

5.1.3.5 *RQ*$_5$. *Toward a Combination of Automated Static Analysis Tools for Code Smell Detection.*

In this research question, we took into account the possibility to devise a combined model that mixes together the outputs of different static analysis tools.

Starting from all warning types of the various tools, we have proceeded as follows. In the first place, we built a new dataset where, for all classes of the systems considered, we reported all the warnings raised by all tools. This step led to the creation of unique dataset that combined all the information mined in the context of our previous research questions. In the second place, we have re-run the *Gain Ratio Feature Evaluation* [257] in order to globally rank the features and discard those that, in such a new combined dataset, did not provide any information gain.

After discarding the irrelevant features, we have followed the same steps as **RQ**$_3$ with the aim of conducting a fair comparison of the combined model with the individual ones previously experimented. As such, we trained the model using multiple classifiers appropriately configured using the MULTISEARCH algorithm [334] and considering the problem of data balancing. Afterwards,

to verify the performance of the combined model, we adopted the same validation strategy as $\mathbf{RQ}_3$ and compared it with the values of F-Measure, and Matthews Correlation Coefficient obtained by the individual models. Finally, we used the Nemenyi test [214] for statistical significance.

5.1.3.6 *RQ_6. Comparison with a baseline machine learner.*

To address $\mathbf{RQ}_6$, we had to first select an existing solution to compare with. Most of the previous studies [5, 15, 132] experimented with various machine learning techniques, yet they all employed code metrics as predictors. As an example, Maiga et al. [178] characterized *God Class* instances by means of Object-Oriented metrics. Similarly, other researchers have attempted to verify how different machine learning algorithms work in the task of code smell classification without focusing on the specific features to use for this purpose [15]. Hence, we decided to devise a baseline machine learning technique that uses code metrics as predictors. In this respect, we computed the entire set of metrics proposed by Chidamber and Kemerer's suite [50] with our own tool and use them as features.

After computing the code metrics, we followed exactly the same methodological procedure used in the context of $\mathbf{RQ}_3$ and $\mathbf{RQ}_5$. As such, the baseline machine learner aimed at predicting the presence/absence of code smells. Also in this case, we experimented with various machine learning algorithms, finding *Random Forest* to be the best one. When training the baseline, we took care of possible multi-collinearity by excluding the code metrics providing no information gain, other than tuning the hyper-parameters by means of the MULTISEARCH algorithm [334]. In terms of data balancing, we verified what was the best possible configuration, benchmarking *Class Balancer*, *Resample*, *Smote*, and *NoBalance*: *Smote* was found to be the best option.

We applied a 10-fold cross validation on the dataset, so that we could have a fair comparison with the approach devised in $\mathbf{RQ}_5$—note that we did not consider a full comparison with the individual models experimented in $\mathbf{RQ}_3$ since these were shown already to be less performing. The accuracy of the baseline was assessed through F-Measure, and MCC. Finally, we executed

the post-hoc Nemenyi test [214] on the distributions of MCC values achieved by the baseline and the combined machine learner output by $\mathbf{RQ}_5$, setting the significance level to 0.05.

5.1.3.7 *RQ_7. Orthogonality between the warning- and metric-based Detection Models.*

In this research question we performed a complementarity analysis between the warning- and the metric-based Detection Models. In order to perform such a complementarity analysis, we followed the same methodology applyed for $\mathbf{RQ}_4$. In particular, for each actual smelly instance, we computed the overlap metrics described in Section 5.1.3.4, i.e., $correct_{m_i \cap m_j}$ and $correct_{m_i \setminus m_j}$.

5.1.3.8 *RQ_8. Combining static analysis warnings and code metrics.*

To study the performance of a machine learner that exploits both static analysis warnings and code metrics, we have proceeded in a similar manner as the other research questions, After combining all the metrics experimented so far in a unique dataset, we re-run the *Gain Ratio Feature Evaluation* [257] to understand the contribution provided by each of those metrics. As previously done, we discarded the ones whose contribution was null. Afterwards, we followed the same steps as $\mathbf{RQ}_5$ and compared the performance of the combined model to the previously built models using F-Measure, and MCC, other than the Nemenyi test [214] for statistical significance.

5.2 ANALYSIS OF THE RESULTS

In the following, we discuss the results achieved when addressing our research questions. For the sake of understandability, we report the discussion by RQ.

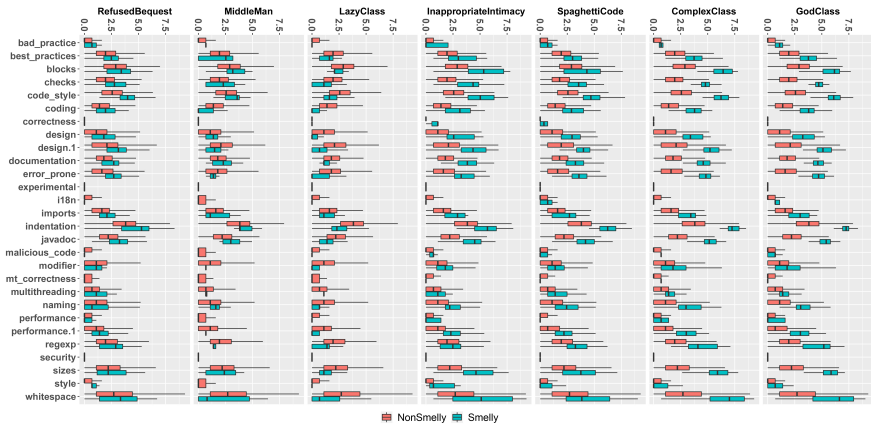

Figure 5.1: Boxplots reporting warnings distributions in smelly/non smelly classes for the seven code smells considered.

5.2.1 RQ_1. *Distribution analysis.*

Figure 5.1 shows boxplots of the distributions of warning categories in smelly and non-smelly classes for the seven code smell types considered in the study. Regardless of the code smell and the warning category considered, the distributions always contain higher values for smelly cases, i.e., smelly classes are more likely to contain a higher number of warnings. The only exception is represented by *Lazy Class*, in which the greater number of warnings arises in classes that are not affected by this code smell. Although this result could sound strange, it is fair to remember that *Lazy Class* refers to very short classes that basically have no responsibility. Therefore, it is reasonable to think that lazy classes are associated with few or no warnings. Table 5.3 reports results for the Mann-Whitney and Cliff's Delta tests. Results indicate that for most of the warning categories, there is a statistically significant difference between the two distributions, thus indicating that those categories represent relevant features to discriminate smelly and non-smelly instances. Turning to the analysis of the categories related to each individual tool, we can see that *PMD* yields the most relevant warnings. Indeed, except for *Middle Man* and *Lazy Class*, all the warning categories belonging to this tool resulted to be relevant. Similarly, *Checkstyle*'s warning categories are very

Table 5.3: Mann Whitney and Cliff's Delta Statistical Test Results. We use N, S, M, and L to indicate negligible, small, medium and large effect size respectively. Significant values are reported in bold.

		God Class		Complex Class		Spaghetti Code		Inapp. Intimacy		Lazy Class		Middle Man		Refused Bequest	
Tool	**Warning**	**p-value**	δ	**p-value**	δ	**p-value**	δ	**p-value**	δ	**p-value**	δ	**p-value**	δ	**p-value**	δ
Checkstyle	regexp	**3.2e-68**	**M**	**9.9e-66**	**M**	4.1e-02	N	3.1e-04	N	2.5e-01	N	**8.7e-08**	**S**	**9.9e-06**	**N**
	checks	**1.6e-86**	**L**	**1.7e-57**	**L**	**3.3e-13**	**N**	**4.2e-23**	**M**	**1.8e-08**	**S**	**1.7e-04**	**S**	**1e-15**	**S**
	whitespace	**3e-93**	**L**	**1.6e-69**	**L**	**2.6e-17**	**S**	**1e-25**	**M**	8.5e-01	N	**4.6e-05**	**S**	**1.1e-15**	**S**
	blocks	**1.5e-100**	**L**	**3.8e-68**	**L**	**1.2e-20**	**S**	**1.6e-36**	**M**	7.7e-01	N	**3.3e-18**	**L**	**1.2e-18**	**S**
	sizes	**3.2e-77**	**L**	**9.7e-50**	**L**	**1.7e-04**	**N**	**4.9e-23**	**M**	8.7e-01	N	7.4e-01	N	6.4e-02	N
	javadoc	**2.2e-74**	**L**	**3.8e-46**	**L**	**1.4e-10**	**N**	**3.8e-23**	**M**	7e-04	S	**1e-09**	**M**	**2.2e-10**	**S**
	indentation	**3.1e-60**	**M**	**1e-38**	**M**	**1.1e-12**	**N**	**2.6e-15**	**S**	5.2e-03	N	1.7e-04	S	2.1e-04	N
	naming	**1.4e-128**	**L**	**2.8e-78**	**L**	**4.8e-39**	**S**	**2.3e-29**	**M**	3.7e-02	N	9.9e-01	N	**2.8e-11**	**N**
	imports	**1.1e-40**	**M**	**5.7e-27**	**M**	3.3e-02	N	**4.2e-22**	**M**	7.5e-02	N	5.8e-01	N	**4.6e-06**	**N**
	coding	**2.2e-114**	**L**	**2.3e-77**	**L**	**2e-43**	**S**	**1.2e-35**	**M**	1.7e-01	N	1.8e-01	N	**5.8e-08**	**N**
	design	**1.2e-68**	**M**	**1.5e-39**	**M**	**2.5e-11**	**N**	**1e-23**	**M**	3.8e-03	N	**5.8e-12**	**M**	**3.4e-05**	**N**
	modifier	**6e-136**	**M**	**4.9e-103**	**M**	**1.9e-17**	**N**	**1.3e-47**	**S**	8.1e-01	N	3.4e-01	N	1.5e-01	N
Findbugs	style	**1.1e-63**	**S**	**7.9e-20**	**N**	**2.2e-120**	**S**	**4.2e-19**	**N**	4.9e-01	N	7.3e-02	N	**9.2e-07**	**N**
	correctness	**2e-07**	**N**	1.7e-02	N	**4.1e-25**	**N**	4.7e-02	N	6.1e-01	N	5.6e-01	N	1.3e-01	N
	performance	**1.2e-13**	**N**	**2.5e-19**	**N**	**2.5e-23**	**N**	**1.5e-37**	**N**	9.6e-01	N	2.8e-01	N	**8.2e-07**	**N**
	malicious_code	1.1e-04	N	1.3e-01	N	1.2e-04	N	**8.8e-12**	**N**	5.2e-01	N	3.1e-01	N	4.2e-01	N
	bad_practice	**7.3e-23**	**N**	5.6e-03	N	**2.5e-112**	**N**	**2.4e-36**	**S**	1.3e-01	N	**3.4e-08**	**N**	8.5e-03	N
	i18n	**3.5e-10**	**N**	4e-03	N	**4e-101**	**N**	**8.3e-08**	**N**	4.1e-01	N	2.6e-01	N	1.8e-01	N
	mt_correctness	**2.1e-10**	**N**	3e-01	N	**2.9e-21**	**N**	**4.4e-26**	**N**	5e-01	N	6.1e-01	N	1.9e-01	N
	experimental	5.5e-01	N	6.2e-01	N	**6.4e-18**	**N**	6.6e-01	N	7.4e-01	N	8e-01	N	5.2e-01	N
	security	7.7e-01	N	8.1e-01	N	**1.1e-79**	**N**	8.3e-01	N	8.7e-01	N	9e-01	N	7.5e-01	N
PMD	documentation	**4.1e-233**	**L**	**2.9e-145**	**L**	**1.9e-190**	**L**	**7.7e-70**	**L**	**2.9e-09**	**S**	3.2e-03	S	**4.6e-31**	**S**
	code_style	**6.5e-233**	**L**	**2e-160**	**L**	**1.5e-302**	**L**	**8.3e-73**	**L**	**1.3e-08**	**S**	**2.8e-05**	**S**	**3.3e-79**	**L**
	best_practices	**3.6e-166**	**L**	**3.1e-120**	**L**	**1.3e-210**	**L**	**2e-43**	**L**	9.9e-03	N	8.9e-01	N	**1.2e-66**	**M**
	design	**1.6e-236**	**L**	**1.1e-164**	**L**	**0e+00**	**L**	**1.8e-62**	**L**	**1.3e-06**	**S**	7.4e-01	N	**2e-63**	**M**
	error_prone	**4.2e-239**	**L**	**1.9e-162**	**L**	**0e+00**	**L**	**2.1e-59**	**L**	1.3e-04	S	1.7e-01	N	**3.9e-67**	**M**
	multithreading	**3.7e-177**	**M**	**5.3e-109**	**M**	**4.2e-93**	**S**	**1.3e-22**	**S**	8.9e-01	N	3.6e-01	N	**1.3e-16**	**N**
	performance	**1.2e-285**	**L**	**4.7e-204**	**L**	**0e+00**	**L**	**2.2e-95**	**L**	**5.3e-08**	**S**	6.8e-01	N	**7.5e-62**	**M**

relevant for six out of the seven code smells considered. Finally, the warnings generated by *Findbugs* are those showing the smaller differences between the two considered distributions.

Summing Up: Results of our distribution analysis indicate that warnings generated by Automatic Static Analysis Tools could be good indicator of the presence of code smell instances. While *Checkstyle* and *PMD* generate a wide set of significant warnings, *Findbugs*'s warnings seem to be less correlated with code smells.

5.2.2 RQ_2. *Contribution of static analysis warnings in code smell detection.*

Table 5.4 reports the mean information gain values obtained by the metrics composing the 21 models built in our study. For the sake of readability, we

Table 5.4: Information Gain of our independent variables for each static analysis tool.

	Checkstyle		FindBugs		PMD	
Code Smell	**Metric**	**Mean**	**Metric**	**Mean**	**Metric**	**Mean**
God Class	Indentation	0.03	Style	0.02	Code Style	0.03
	Blocks	0.03	Bad Practice	0.01	Documentation	0.03
	Sizes	0.03	I18N	0.01	Error Prone	0.03
Complex Class	Indentation	0.04	Style	0.02	Code Style	0.03
	Blocks	0.04	Security	0.01	Design	0.03
	Sizes	0.03	Malicious Code	0.01	Error Prone	0.03
Spaghetti Code	Indentation	0.03	I18N	0.01	Error Prone	0.03
	Blocks	0.02	Security	0.01	Code Style	0.03
	Coding	0.02	Correctness	0.01	Design	0.03
Inappropriate Intimacy	Whitespace	0.01	Bad Practice	0.02	Code Style	0.01
	Indentation	0.01	Style	0.01	Error Prone	0.01
	Javadoc	0.01	Correctness	0.01	Design	0.01
Lazy Class	Javadoc	0.01	Security	0.01	Code Style	0.01
	Sizes	0.01	Malicious Code	0.01	Documentation	0.01
	Indentation	0.01	Correctness	0.01	Design	0.01
Middle Man	Indentation	0.01	Security	0.01	Error Prone	0.01
	Design	0.01	Malicious Code	0.01	Documentation	0.01
	Checks	0.01	Correctness	0.01	Code Style	0.01
Refused Bequest	Indentation	0.01	Style	0.01	Code Style	0.01
	Checks	0.01	Security	0.01	Error Prone	0.01
	Design	0.01	Malicious Code	0.01	Design	0.01

just reported the three most relevant warning categories for each model, i.e., one for each tool-smell combination.

Looking at the achieved results, the first thing to notice is that, depending on the code smell type, the warning types could have different weights: this practically means that a machine learner for code smell identification should exploit different features depending on the target code smell rather than rely on a unique set of metrics to detect them all. As an example, the *Indentation* type of CHECKSTYLE provides different information gain based on the specific code smell type. This seems to suggest that not all warnings would have the same impact on the performance of various code smell detectors.

When analyzing the most powerful features of CHECKSTYLE and PMD, we could notice that features related to source code readability are constantly at the top of the ranked list for all the considered code smells. This is, for instance, the case of the *Indentation* warnings given by CHECKSTYLE or

the *Code Style* metrics highlighted by PMD. The most relevant warnings also seem to be strongly related to specific code smells: as an example, the presence of a high number of blocks having a large size might strongly affect the likelihood to have a *God Class* or or a *Complex Class* smell; similarly, design-related issues are the most characterizing aspects of a *Spaghetti Code* or a *Middle Man*. In other words, from this analysis, we could delineate a relation between the most relevant warnings highlighted by CHECKSTYLE and PMD and the specific code smells considered in this study.

A different discussion should be done for FINDBUGS: in this case, the most powerful metrics mostly relate to *Performance* or *Security*, which are supposed to cover different code issues than code smells. As such, we expect this static analysis tool to have lower performance when used for code smell detection.

Finally, it is worth noting that the information gain of the considered features seems to be generally low. On the one hand, this may potentially imply a low capability of the features when employed within a machine learning model. On the other hand, it may also be the case that such a little information would already be enough to characterize and predict the existence of code smell instances. The next sections address this point further.

Summing Up: Generally, the considered features provide low information gain. The most relevant features are related to readability issues when relying on the models built on top of CHECKSTYLE and PMD (e.g., *Indentation*, *Code Style*). As for FINDBUGS, the most relevant features relate to other non functional aspects, e.g., *Performance*, *Security*.

Table 5.5: Aggregate results reporting the performance of the models built with the warning generated by the three static automatic tools.

	Checkstyle				FindBugs				PMD			
	Prec.	**Recall**	**FM**	**MCC**	**Prec.**	**Recall**	**FM**	**MCC**	**Prec.**	**Recall**	**FM**	**MCC**
God Class	0.01	0.62	0.02	0.04	0.01	0.25	0.01	0.01	0.43	0.52	0.47	0.47
Complex Class	0.01	0.48	0.01	0.02	0.00	0.22	0.01	0.00	0.28	0.35	0.31	0.31
Spaghetti Code	0.02	0.43	0.03	0.05	0.01	0.19	0.02	0.00	0.26	0.22	0.24	0.23
Inappropriate Intimacy	0.01	0.44	0.01	0.03	0.00	0.31	0.00	-0.01	0.08	0.17	0.11	0.11
Lazy Class	0.01	0.13	0.01	0.02	0.00	0.63	0.00	-0.01	0.04	0.11	0.06	0.06
Middle Man	0.00	0.15	0.00	-0.02	0.00	0.66	0.00	0.01	0.08	0.03	0.04	0.05
Refused Bequest	0.01	0.38	0.01	0.00	0.01	0.50	0.01	0.00	0.27	0.14	0.18	0.19

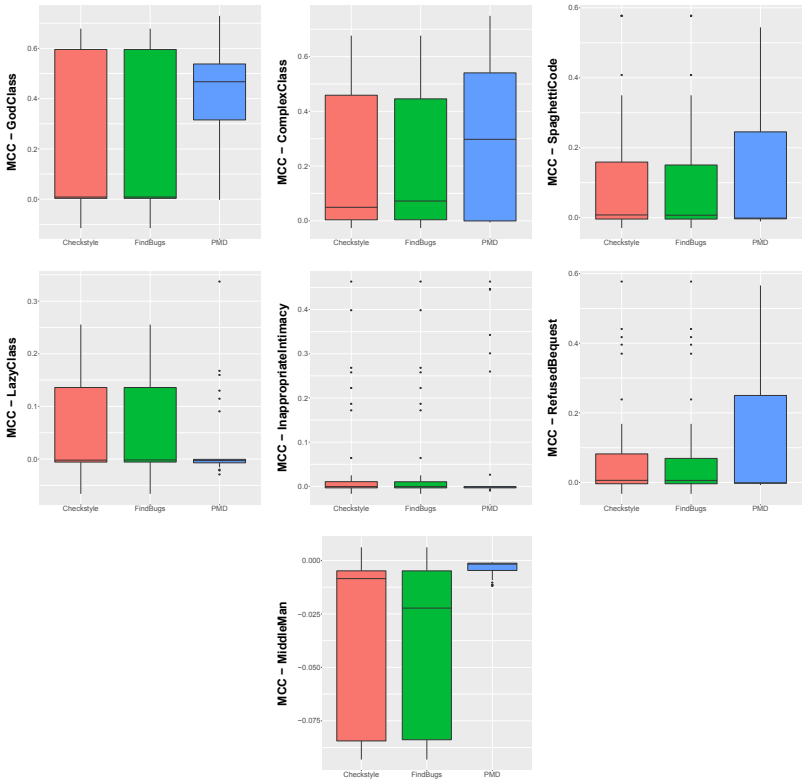

Figure 5.2: Boxplots representing the MCC values obtained by Random Forest trained on static analysis warnings for code smells detection.

5.2.3 RQ_3. *The role of static analysis warnings in code smell detection.*

Figure 5.2 reports the performance capabilities in terms of MCC of the models built using the warnings given by CHECKSTYLE, FINDBUGS, and PMD, respectively. In this section, we only discuss the overall results obtained with the best configuration of the models, namely the one considering *Random Forest* as classifier and *Class Balancer* as data balancing algorithm.

We can immediately point out that the models built using the warnings of static analysis tools have very low performance. In almost all cases, indeed, the MCCs show median values that are very close to zero, indicating a very low, if not even null correlation between the set of detected and the set

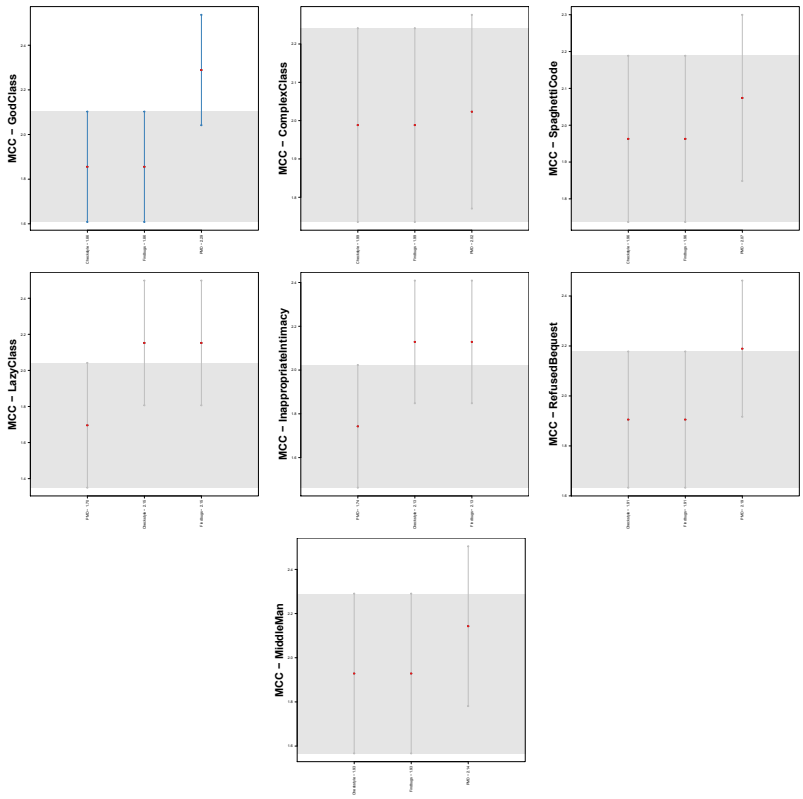

Figure 5.3: Plots representing the results of Nemenyi test for statistical significance between the MCC values obtained by Random Forest trained on static analysis warnings for code smells detection.

of actual smelly instances. This result is in line with previous studies on the application of machine learning for code smell detection [71]. As an example, we previously reported that models built using code metrics of the Chidamber-Kemerer suite [50] work worst than a constant classifier that always considers an instance as non-smelly.

The reasons behind the low MCC values could be various. This coefficient is computed by combining true positives, true negatives, false positives, and false negatives altogether; as such, having a clear understanding of the factors impacting those values is not trivial. In an effort of determining these reasons,

Table 5.5 provides a more detailed overview of the performance of the models for each of the considered tools and code smells.

The first aspect to consider is that, when considering CHECKSTYLE and FINDBUGS, the low performance could be due to the high false-positive rate. Indeed, despite the moderately high recall, the results are negatively influenced by the very low precision that is always close to zero. A different conclusion must be drawn for PMD. The results show similar precision and recall values when considering the code smells individually, but these values are higher or lower depending on the specific code smell type. In other words, our results indicate that the models built using the warnings provided by this tool could achieve higher or lower performance, depending on the smell considered—hence, the capabilities of these models cannot be generalized to all code smells.

Another important aspect to take into account is the different behaviour of the three models with respect to the code smell to detect. While CHECKSTYLE and PMD achieve better performance in detecting *God Class*, *Complex Class*, and *Spaghetti Code*, FINDBUGS gives its best in the detection of *Lazy Class*, *Middle Man*, and *Refused Bequest*.

Figure 5.3 confirms the discussion above. Indeed, by analyzing the statistical difference between models with respect to code smells, we can notice that PMD performance are statistically better than the other two models when detecting *God Class* instances. In the cases of *Lazy Class* and *Inappropriate Intimacy* code smells, instead, models built with warning generated by CHECKSTYLE, and FINDBUGS performs significantly better than those relying on PMD warnings.

Nonetheless, despite the negative results achieved so far, it is worth reflecting on two specific aspects coming from our analysis. On the one hand, for each code smell there is at least one tool whose warnings are able to catch a good number of smelly instances (i.e., recall $\approx 50\%$). On the other hand, different warning categories achieve higher performance on different sets of code smells. Based on these two considerations, we conjectured that higher performance could be potentially achieved when combining the warnings

generated by the three static analysis tools. Next paragraphs address this point deeply.

Summing Up: Machine-Learning based code smell detection approaches using static analysis warning as independent variables generally achieve low performance. Specifically, in many cases, those approaches achieve a good recall but a very bad precision, indicating a high false-positive rate. Differences in the performance achieved by the three warning categories with respect to the code smell analyzed could indicate that a combination of these categories could help achieving higher performance.

Table 5.6: Overlap analysis between Checkstyle and Findbugs.

Code Smell	CS ∩ FB	CS \ FB	FB \ CS
God Class	7%	47%	46%
Complex Class	11%	37%	52%
Spaghetti Code	5%	70%	25%
Inappropriate Intimacy	8%	23%	69%
Lazy Class	0%	7%	93%
Middle Man	8%	0%	92%
Refused Bequest	21%	25%	54%

5.2.4 *RQ_4. Orthogonality of the Prediction Models.*

In the context of the fourth research question, we sought to move toward a combination of warning types coming from different static analysis tools for code smell detection. Let discuss the results by analyzing Table 5.6, that reports the overlap between the model using the warnings generated by CHECKSTYLE and the one built on the FINDBUGS warnings. It is interesting to observe that there is a very high complementarity between the two models, regardless on the code smell considered. Indeed, only a small portion of smelly instances are correctly identified by both the models, i.e., $(CS \cap FB) \leq$

21%. Moreover, the percentage of instances correctly classified by only one of the models is generally high, indicating such complementarity.

Table 5.7: Overlap analysis between Checkstyle and PMD.

Code Smell	CS ∩ PMD	CS \ PMD	PMD \ CS
God Class	0%	98%	2%
Complex Class	0%	98%	2%
Spaghetti Code	2%	94%	4%
Inappropriate Intimacy	33%	60%	7%
Lazy Class	0%	100%	0%
Middle Man	0%	100%	0%
Refused Bequest	0%	100%	0%

Table 5.7 show the results of the overlap between the models built on CHECKSTYLE and PMD warnings. The table immediately suggests that PMD provides a very limited contribution in terms of smelly instances detected. Results suggest that for almost all code smells, CHECKSTYLE alone could achieve the same results, if not even better, of a possible combination of the two tools.

Table 5.8: Overlap analysis between Findbugs and PMD.

Code Smell	FB ∩ PMD	FB \ PMD	PMD \ FB
God Class	1%	98%	1%
Complex Class	0%	98%	2%
Spaghetti Code	2%	87%	11%
Inappropriate Intimacy	10%	84%	6%
Lazy Class	0%	100%	0%
Middle Man	0%	100%	0%
Refused Bequest	0%	100%	0%

Table 5.8 provides the overlap results for FINDBUGS and PMD. These results deserve a discussion similar to the previous one. Indeed, as we discussed above, also in this case PMD does not provide an important contribution.

Most of the correctly classified instances are indeed provided by the model built only on FINDBUGS warnings.

Table 5.9: Overlap Analysis considering each tool independently.

Code Smell	CS \ (FB ∪ PMD)	FB \ (CS ∪ PMD)	PMD \ (CS ∪ FB)	CS ∩ FB ∩ PMD
God Class	44%	56%	0%	0%
Complex Class	38%	59%	2%	0%
Spaghetti Code	74%	23%	2%	1%
Inappropriate Intimacy	40%	46%	1%	13%
Lazy Class	4%	95%	1%	0%
Middle Man	21%	79%	0%	0%
Refused Bequest	36%	62%	2%	0%

Finally, looking at the overlap results for all the three models, shown in Table 5.9, we can confirm the above results. The low percentage of instances that are simultaneously correctly detected as smelly by all three approaches indicates a high complementarity between the instances detected by the three tools, i.e., different tools are able to detect different sets of smelly instances. Such complementarity is an indicator that better performance could be achieved by combining the warnings generated by the three tools in a unique, unified, detection model.

Summing Up: Machine Learning code smell detection models built on the warning generated by different tools are highly complementary. Both CHECKSTYLE and FINDBUGS are able to identify a great number of instances that are not detected by the other. PMD detects instances undiscovered by the others only in a limited number of cases.

5.2.5 RQ_5. *Toward a Combination of Automated Static Analysis Tools for Code Smell Detection.*

In the context of this **RQ**, we defined and evaluated a combined model. As explained in Section 5.2.2, we faced the problem by first measuring the potential information gain by the warning types when put all together and then considering the most relevant warnings for the definition of a more effective combination.

Table 5.10: Information Gain of our independent variables for the combined model.

	Combined model	
Code Smell	**Metric**	**Mean**
God Class	Code.Style	0.03
	Documentation	0.02
	Design	0.02
Complex Class	Code Style	0.03
	Design	0.02
	Error Prone	0.02
Spaghetti Code	Error Prone	0.03
	Code Style	0.02
	Design	0.02
Inappropriate Intimacy	Code Style	0.01
	Whitespace	0.01
	Design	0.01
Lazy Class	Javadoc	0.01
	Sizes	0.01
	Code Style	0.01
Middle Man	Imports	0.01
	Design	0.01
	Checks	0.01
Refused Bequest	Code Style	0.01
	Error Prone	0.01
	Documentation	0.01

Table 5.10 reports the information gain values obtained by the metrics composing the combined models. Also in this case, for the sake of readability we only reported the three most relevant categories for each model.

Looking at the table, the first consideration we can do is that readability-related features remain relevant even when considering all the features together. Some examples are *Code Style* for *God Class* or *Javadoc* for *Lazy Class*. Differently, features related to performance and security aspects, that have been shown to be relevant in the models built only on FINDBUGS warnings, are no longer important when combining the tools.

Another important aspect is related to the presence of design-related features in the list of the most relevant predictors. Those features, that are the more in-line with the definition of code smell, were surprisingly excluded in the context of our $\mathbf{RQ}_2$. The fact that they become more relevant when the three tools are combined may represent an indicator of the fact that a combined model can outperform the models discussed in $\mathbf{RQ}_3$.

Table 5.11: Results reporting the performance of the model built by combining the warning generated by the three static automatic tools.

	Checkstyle				FindBugs				PMD				Combined			
	Prec.	Recall	FM	MCC	Prec.	Recall	FM	MCC	Prec.	Recall	FM	MCC	Prec.	Recall	FM	MCC
God Class	0.01	0.62	0.02	0.04	0.01	0.25	0.01	0.01	0.43	0.52	0.47	0.47	0.49	0.47	0.48	0.48
Complex Class	0.01	0.48	0.01	0.02	0.00	0.22	0.01	0.00	0.28	0.35	0.31	0.31	0.34	0.34	0.34	0.34
Spaghetti Code	0.02	0.43	0.03	0.05	0.01	0.19	0.02	0.00	0.26	0.22	0.24	0.23	0.31	0.19	0.24	0.24
Inappropriate Intimacy	0.01	0.44	0.01	0.03	0.00	0.31	0.00	-0.01	0.08	0.17	0.11	0.11	0.21	0.15	0.17	0.17
Lazy Class	0.01	0.13	0.01	0.02	0.00	0.63	0.00	-0.01	0.04	0.11	0.06	0.06	0.17	0.12	0.14	0.14
Middle Man	0.00	0.15	0.00	-0.02	0.00	0.66	0.00	0.01	0.08	0.03	0.04	0.05	0.56	0.07	0.13	0.20
Refused Bequest	0.01	0.38	0.01	0.00	0.01	0.50	0.01	0.00	0.27	0.14	0.18	0.19	0.39	0.09	0.15	0.18

Table 5.11 and Figure 5.4 show the performance of the combined model. As we can see, there is a general improvement, particularly in terms of precision—hence confirming our hypothesis on the potential of combining features of different static analysis tools to reduce false positives. The MCC values, ranging between 14% and 48% are clearly better than the one provided by the single models, as discussed in $\mathbf{RQ}_3$. Results of Nemenyi test, reported in Figure 5.5, evidenced a clear statistical difference between the MCCs achieved by the combined model and the ones provided by single-tool models. However, unfortunately, these results still indicate the unsuitability of machine learning approaches for code smell detection, as already proven in previous studies in the field [71].

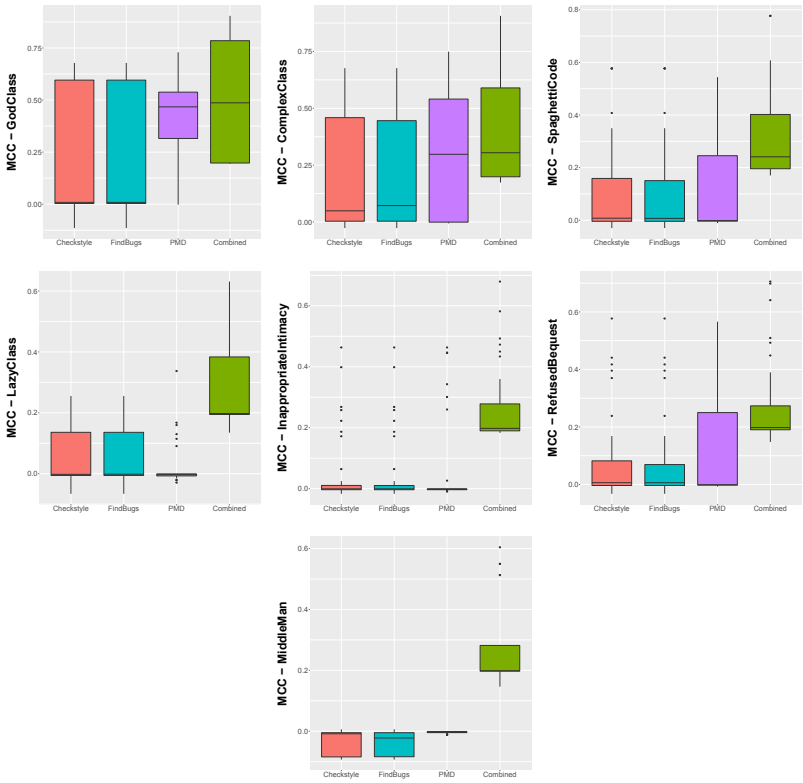

Figure 5.4: Boxplots representing the MCC values obtained by Random Forest trained on static analysis warnings for code smells detection.

Summing Up: Design-related features become important when the tool's warnings are combined. The combined model outperforms the three models described in **RQ**$_3$. However, the overall performance is still quite low, reinforcing past findings about the unsuitability of ML-based code smell detection approaches.

5.2.6 *RQ*$_6$. *Comparison with a baseline machine learner.*

Table 5.12 and Figure 5.6 report the results regarding the comparison of the performance achieved by the model that uses the combination of the warnings

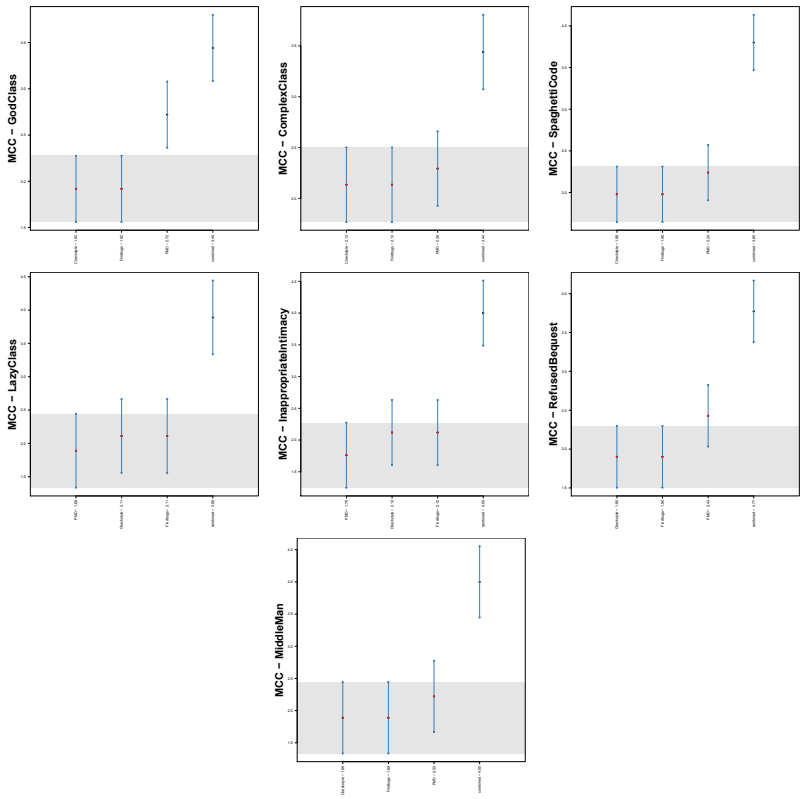

Figure 5.5: Plots representing the results of Nemenyi test for statistical significance between the MCC values obtained by Random Forest trained on static analysis warnings for code smells detection.

generated by the three ASATs considered, and the model using structural information as predictors. The first consideration is that the model using the warnings generated by the three ASATs seems to slightly outperform the model using structural information for almost all the code smell types. In particular, this is the case of *Lazy Class*, *Inappropriate Intimacy*, *Refused Bequest*, and *Middle Man*. These four smells do not have a direct correlation with structural information given to the structural classifier. For instance, while we can use simple structural metrics such as size and complexity to identify *God Class* and *Spaghetti Code* instances, the ML model using

Table 5.12: Aggregate results reporting the comparison of the warning-based model with the metric-based one.

	Warning				Metric			
	Prec.	**Recall**	**FM**	**MCC**	**Prec.**	**Recall**	**FM**	**MCC**
God Class	0.49	0.47	0.48	0.48	0.30	0.83	0.44	0.49
Complex Class	0.34	0.34	0.34	0.34	0.18	0.61	0.27	0.32
Spaghetti Code	0.31	0.19	0.24	0.24	0.15	0.34	0.21	0.22
Inappropriate Intimacy	0.21	0.15	0.17	0.17	0.10	0.23	0.14	0.15
Lazy Class	0.17	0.12	0.14	0.14	0.00	0.00	0.00	0.00
Middle Man	0.56	0.07	0.13	0.20	0.00	0.00	0.00	0.00
Refused Bequest	0.39	0.09	0.15	0.18	0.21	0.02	0.03	0.06

structural information does not include precise metrics describing other aspects such as laziness or intimacy level between classes.

The results of the Nemenyi test depicted in Figure 5.7, confirm that in the cases described above there is a statistically significant difference in the two distributions. On the other hand, with respect to *God Class*, and *Spaghetti Code* it is not possible to clearly establish which of the models perform better.

Summing Up: The ML model using ASATs warnings and the one using structural information achieve very similar performance in detecting code smells whose definition is strictly correlated with the structural information involved. In all the other cases, the model using warning categories as predictors appears to have better detection capabilities than the one using only structural information.

5.2.7 RQ_7. *Orthogonality between the warning- and metric-based Detection Models.*

Table 5.13 reports results of the complementarity analysis conducted between the warning- and the metric-based machine learning detection models. The most evident result is that, regardless of the code smell considered, the two techniques show a strong overlap, i.e., most of the smelly instances identified by a technique are also identified by the other. Such a strong overlap could indicate that using metrics and warnings in combination would

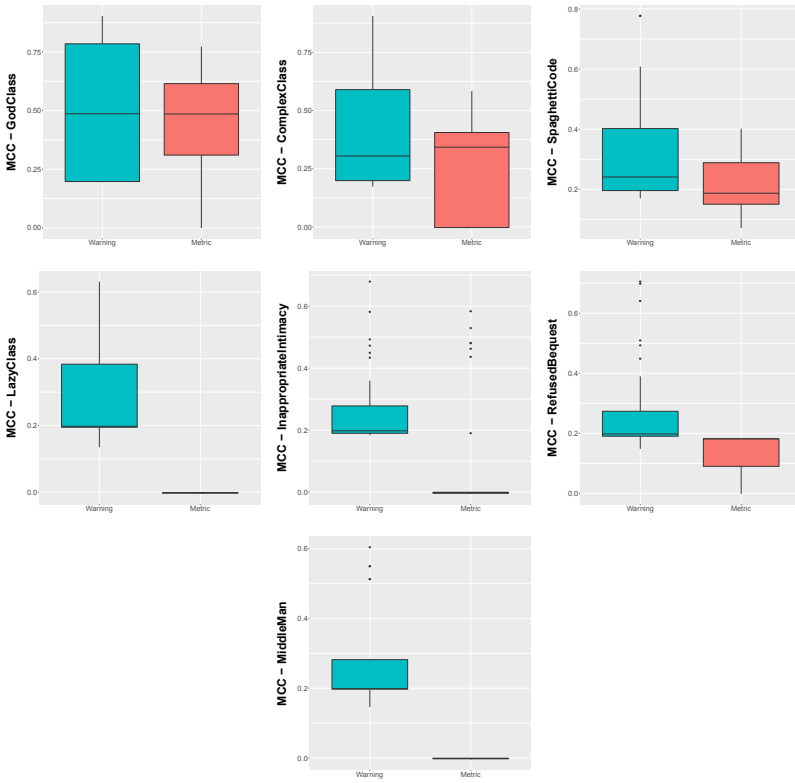

Figure 5.6: Boxplots representing the MCC values obtained by Random Forest trained on static analysis warnings and structural metrics for code smells detection.

not lead to performance improvements. This is particularly true for *Lazy Class*, *Refused Bequest*, and *Middle Man* for which there is a very small complementarity. However, as for *God Class*, *Complex Class*, *Spaghetti Code*, and *Inappropriate Intimacy*, results show that there exist a number of smelly instances that only one of the techniques is able to detect, thus indicating a complementarity, even if limited. Therefore, it could be still worth to assess the performance achieved by a machine learner based on both warnings and structural metrics.

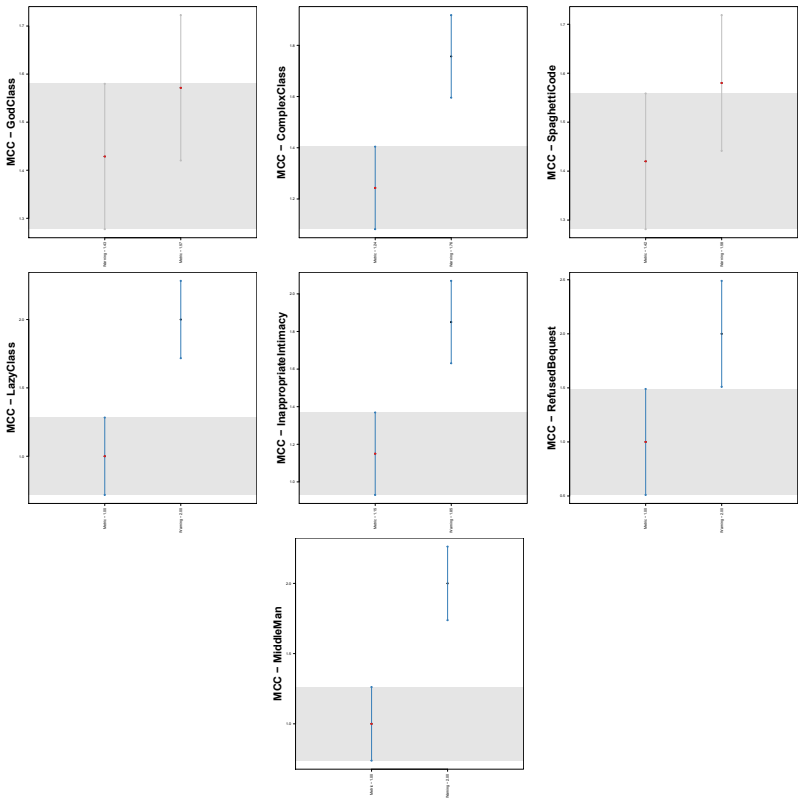

Figure 5.7: Plots representing the results of Nemenyi test for statistical significance between the MCC values obtained by Random Forest trained on static analysis warnings and structural metrics for code smells detection.

Summing Up: The warning- and the metric-based machine learning code smell detection models have a strong overlap, regardless of the smell considered. However, since in some cases the results showed a complementarity, although limited, we think that a combination of these two set of predictors could still lead to a performance improvement.

5.2.8 RQ_8. *Combining static analysis warnings and code metrics.*

Table 5.14 and Figure 5.8 report the results of the performance achieved by the two model based only on ASATs warnings and code metrics, and

Table 5.13: Overlap analysis between the warning- and metric-based Detection Models.

Code Smell	Warning ∩ Metric	Warning \ Metric	Metric \ Warning
God Class	81%	11%	6%
Complex Class	76%	16%	8%
Spaghetti Code	72%	18%	10%
Inappropriate Intimacy	64%	22%	22%
Lazy Class	98%	1%	1%
Middle Man	86%	9%	5%
Refused Bequest	89%	7%	4%

Table 5.14: Aggregate results reporting the comparison of the combined model with the model combining warnings categories and structural metrics.

	Warning				Metric				Combined			
	Prec.	Recall	FM	MCC	Prec.	Recall	FM	MCC	Prec.	Recall	FM	MCC
God Class	0.49	0.47	0.48	0.48	0.30	0.83	0.44	0.49	0.53	0.58	0.56	0.55
Complex Class	0.34	0.34	0.34	0.34	0.18	0.61	0.27	0.32	0.39	0.43	0.41	0.41
Spaghetti Code	0.31	0.19	0.24	0.24	0.15	0.34	0.21	0.22	0.36	0.21	0.25	0.27
Inappropriate Intimacy	0.21	0.15	0.17	0.17	0.10	0.23	0.14	0.15	0.08	0.09	0.10	0.11
Lazy Class	0.17	0.12	0.14	0.14	0.00	0.00	0.00	0.00	0.19	0.12	0.15	0.15
Middle Man	0.56	0.07	0.13	0.20	0.00	0.00	0.00	0.00	0.17	0.06	0.10	0.13
Refused Bequest	0.39	0.09	0.15	0.18	0.21	0.02	0.03	0.06	0.34	0.14	0.20	0.21

the one combining warnings and structural information. Regardless of the considered code smell type, the full model, i.e., the one considering both warnings and structural metrics, appears to slightly outperform the other two. This is particularly true for *God Class*, *Complex Class*, *Spaghetti Code*, and *Inappropriate Intimacy*.

Nemenyi test results, reported in Figure 5.9, confirm that for *God Class*, *Complex Class*, and *Inappropriate Intimacy* the full model performs significantly better than the others. This result is in line with $\mathbf{RQ}_7$ findings. Indeed, a higher complementarity has been shown for such smells, therefore the combined model is able to significantly improve the performance of warning- and metric-based machine learners.

The reported results clearly indicate that adding more information to ML classifiers helps to improve the overall performance in most cases. However, on the other hand, there is still the need of defining a set of metrics that

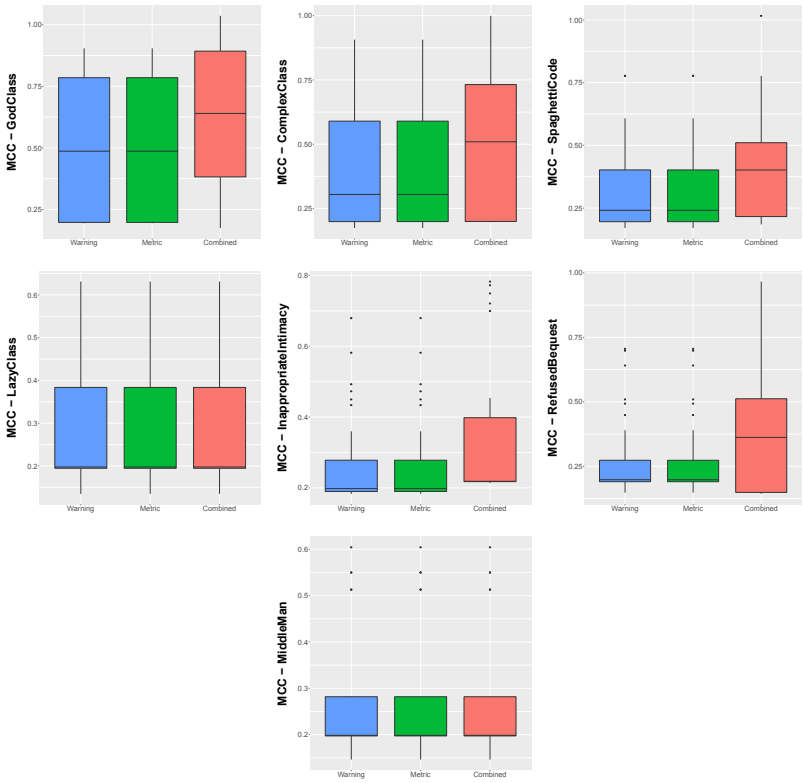

Figure 5.8: Boxplots representing the MCC values obtained by Random Forest trained on static analysis warnings and on the combination of static analysis warnings with structural metrics for code smells detection.

could further improve code smell detection techniques' performance. Our suggestion for future studies is to involve a wider set of predictors of various kinds (e.g., structural, textual, historical) in order to give the classifiers as much information as possible.

Summing Up: The model combining warning categories and structural information significantly outperforms the one based only on ASATs warnings in most of the cases. Adding other metrics to the model could be a winning strategy for future improvements.

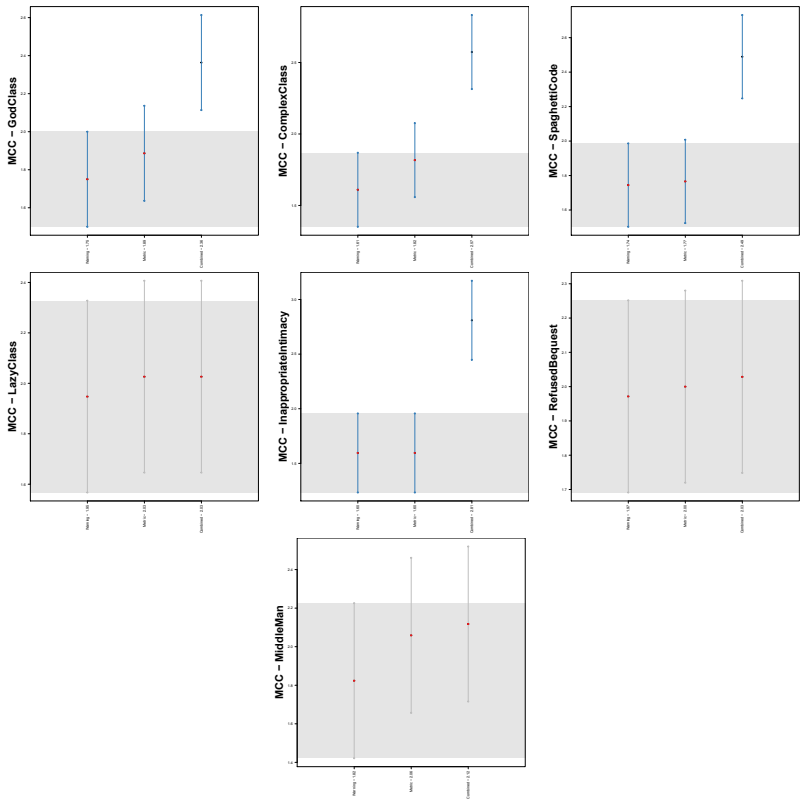

Figure 5.9: Plots representing the results of Nemenyi test for statistical significance between the MCC values obtained by Random Forest trained on static analysis warnings and on the combination of static analysis warnings with structural metrics for code smells detection.

5.3 CONCLUSION

In this chapter, we assessed the adequacy of static analysis warnings in the context of code smell detection. We started by analyzing the contribution given by each warning type to the detection of seven code smell types. Then, we measured the performance of machine learning models using static analysis warnings as features and aiming at identifying the presence of code smells.

The results achieved when experimenting the individual models revealed low performance: this was mainly due to their poor precision. In an effort of dealing with such low performance, we considered the possibility to combine the warnings raised by different static analysis tools: in this regard, we first measured the orthogonality of the code smell instances correctly identified by machine learners exploiting different warnings; then, we combined these warnings in a combined model.

The results of our study reported that, while a combined model can significantly improve the performance of the individual models, it yields a similar accuracy than the one of a random classifier. We also found out that machine learning models built using static analysis warnings reach a particularly low accuracy when considering code smells targeting coupling and inheritance properties of source code.

6

DEVELOPER-DRIVEN CODE SMELL PRIORITIZATION

This chapter presents the first step toward the concept of *developer-driven code smell prioritization* as an alternative and more pragmatic solution to the problem of code smell detection: Rather than ranking code smell instances based on their severity computed using software metrics, we propose to prioritize them according to the criticality perceived by developers.

In particular, we first perform surveys to collect a dataset composed of developers' perception of the severity of 1,332 code smell instances—pertaining to four different types of design problems—for which original developers rated their actual criticality and then propose a novel supervised approach that learns from such labeled data to rank unseen code smell instances. Afterwards, we conduct an empirical study to (1) verify the performance of our prioritization approach, (2) understand what are the features that contribute most to model the developer's perceived criticality of code smells, and (3) compare our approach with the state-of-the-art baseline proposed by Arcelli Fontana and Zanoni [86]. The key differences between our approach and the baseline considered for the comparison are (i) the usage of different kinds of predictors (e.g., process metrics) rather than considering only structural ones, and (ii) the definition of a dependent variable based on the developers' perception.

To sum up, this chapter provides the following contributions:

1. A new dataset reporting the criticality perceived by original developers with respect to 1,332 code smell instances of four different types, which can be further used by the community to build upon our research;

2. The first machine learning-based approach to prioritize code smells according to the real developers' perceived criticality;

3. An empirical study that showcases the performance of our approach, the importance of the employed features, and the reasons why our technique goes beyond the state-of-the-art;

6.1 DATASET CONSTRUCTION

To perform our empirical study, we needed to collect a dataset reporting the perceived criticality of a set of code smells large enough to train a machine learning model. To this aim, we first defined the *objects* of the study, namely (1) a set of software projects and (2) the code smell types we were interested in with their corresponding detectors; Then, we inquired the *subjects* of our study, namely the original developers of the considered projects, in order to collect their perceived criticality of the code smell instances detected on their codebase. The next subsections describe the various steps we followed to build our dataset.

6.1.1 *Selecting projects*

The *context* of the study consisted of nine open-source projects belonging to two major ecosystems such as APACHE[1] and ECLIPSE.[2] Basic information and statistics about the selected projects are summarized in Table 6.1. Specifically, for each considered project, we report (i) the total number of commits available in its change history, (ii) the total number of contributors, and (iii) the size as the number of classes and KLOCs. The selection of these projects was driven by a number of factors. In the first place, we only focused on open-source projects since we needed to access source code to detect the considered design flaws. Similarly, we limited ourselves to JAVA systems as most of the smells, as well as code smell detectors, have been only defined for this programming language [15, 229, 249]. Furthermore, we aimed at analyzing projects having different (a) codebase size, (b) domain, (c) longevity, (d) activity, and (e) population. As such, starting from the set of $2,576$ open-source systems

1 https://www.apache.org
2 https://www.eclipse.org/org/

Table 6.1: Software Projects in Our Dataset.

Project	#Commits	#Devs	#Classes	KLOCs
Apache Mahout	3,054	55	813	204
Apache Cassandra	2,026	128	586	111
Apache Lucene	3,784	62	5,506	142
Apache Cayenne	3,472	21	2,854	542
Apache Pig	2,432	24	826	372
Apache Jackrabbit	2,924	22	872	527
Apache Jena	1,489	38	663	231
Eclipse CDT	5,961	31	1,415	249
Eclipse CFX	2,276	21	655	106
Overall	**32,889**	**436**	**5,506**	**542**

written in JAVA and belonging to the two considered ecosystems available at the time of the analysis on GITHUB,[3] we only took into account those having a number of classes higher than 500, with a change history at least 5 years long, having at least 1,000 commits, and with a number of contributors higher than 20. This filter gave us a total of 682 systems: of these, we randomly selected 9 of them.

6.1.2 *Selecting code smells*

We focused on four class-level types of code smells, namely **Blob (or God Class)**, **Complex Class**, **Spaghetti Code**, and **Shotgun Surgery** (see Table 2.1 for descriptions.

Three specific factors drove the selection of these four code smell types. First, they have been shown to be highly diffused in real software systems [226], thus allowing us to target code smells that are relevant in practice. Second, they are reported to negatively impact maintainability, comprehensibility, and/or testability of software systems [104, 137, 226]: as such, we could investigate design flaws that practitioners may be more able to analyze and assess. Finally, previous findings [227, 288, 330] showed not only that they are

3 https://github.com

actual problems from the developer's perspective, but also that their criticality can be accurately assessed by practitioners, thus mitigating potential problems due to the presence of the so-called conceptual false positives [83], i.e., code smell instances detected as such by automated tools but not representing issues for developers.

6.1.3 *Selecting code smell detectors*

Once we had selected the specific code smells object of our investigation, we then proceeded with the choice of automated code smell detectors that could identify them. Among all the available solutions proposed so far by researchers [78], we opted for DECOR [197] and HIST [228]. The first was selected to identify instances of *Blob*, *Complex Class*, and *Spaghetti Code*, while the latter for the detection of *Shotgun Surgery*.

More specifically, DECOR is an automated solution which adopts a set of "rule cards",[4] namely rules able to describe the intrinsic characteristics that a class must have to be affected by a certain code smell type. In the case of *Blob*, the approach identifies it when a class has a Lack of Cohesion of Method (LCOM5) [122] higher than α, a total number of methods and attributes higher than β, it is associated to many data classes (i.e., classes having just `get` and `set` methods), and has a name having a suffix in the set $\{$*Process, Control, Command, Manage, Drive, System*$\}$, where α and β are relative threshold values. When detecting *Complex Class* instances, DECOR computes the Weighted Methods per Class metric (WMC), i.e., the sum of the cyclomatic complexity of all methods of the class [191], and marks a class as smelly if the WMC is higher than a defined threshold. Finally, *Spaghetti Code* instances are represented by classes presenting (i) at least one method without parameters and having a number of lines of code higher than a defined threshold, (ii) no inheritance, as indicated by the Depth of Inheritance Tree metric (DIT) [50] which must be equal to 1, and (iii) a name suggesting procedural programming, thus having as prefix/suffix a word in the set $\{$*Make, Create, Exec*$\}$. There are two key reasons leading us

4 http://www.ptidej.net/research/designsmells/

to rely on DECOR for the detection of these three smells. In the first place, this detector has been employed in several previous studies on code smells [131, 138, 179, 236, 238], showing good results when considering both precision and recall. At the same time, it implements a lightweight mechanism with respect to other existing approaches (e.g., textual-based techniques relying on information retrieval [234, 241]): the scalability of DECOR allows us to perform an efficient detection on the large systems considered in the study.

Turning our attention to the detection of *Shotgun Surgery*, the discussion is different. Approaches based on source code analysis are poorly effective for the detection of this smell [228]; for instance, the approach proposed by Rao and Reddy [263]—which computes coupling metrics to build a change probability matrix that is then filtered to detect the smell—was not able to identify any *Shotgun Surgery* instance when applied in large-scale studies [228]. For this reason, we relied on HIST [228], a historical-based technique that (1) computes association rules [4] to identify methods of different classes that often change together and (2) identifies instances of the smell if a class contains at least one method that frequently changes with methods present in more than 3 different classes. Such a historical-based approach has shown an F-Measure close to 92% [228], thus representing a suitable solution for conducting our study.

On a technical note, we relied on the original implementations of DECOR and HIST, hence avoiding potential threats to construct validity due to re-implementations.

6.1.4 *Collecting the criticality of code smells*

The last step required to build our dataset was the actual detection of code smells in the considered systems and the subsequent inquiry on their criticality. We followed a similar strategy as Silva *et al.*[273]: in short, in a time period of 6 months, from January 1st to June 30th, 2018, we monitored the activities performed on the selected repositories and, as soon as a developer committed changes to classes affected by any of the considered smells, we sent an e-mail to that developer to ask (i) whether s/he actually perceived/recognized the

presence of a code smell and (ii) if so, rate its criticality using a Likert scale from 1 (very low) to 5 (very high) [163].

In particular, we built an automated mechanism that fetched—using the `git-fetch` command—commits from the repositories to a local copy on a daily basis. This gave us the possibility to generate the list of classes modified during the workday. At this point, we performed the actual smell detection. In the case of DECOR, we parsed each modified class and run the detection rules described in Section 6.1.3 to identify instances of *Blob*, *Complex Class*, and *Spaghetti Code*. As for HIST, it requires information about the change history of the involved classes: for this reason, before running the *Shotgun Surgery* detection algorithm, we mined the log file of the projects to retrieve the set of changes applied on the classes modified during the workday. Through the procedure described so far, we obtained a list of smelly classes, and, for each of them, we stored the e-mail address of the developer who committed changes on it. Afterwards, we manually double-checked the smelly classes given by the automated tools with the aim of discarding possible false positives, thus avoiding asking developers useless information. Overall, the code smell detection phase resulted in a total of $2,675$ candidate code smells. Of these, we discarded 455 ($\approx$17%) false positives.

Finally, we sent e-mails to the original developers. In the text, we first presented ourselves and then explained that our analysis tool suggested that the developer likely worked on a class affected by a design issue—without revealing the exact code smell to avoid confirmation bias [216]. Then we asked three specific questions:

1. *Were you aware of the presence of a design flaw?*

2. *If yes to question (1), may you please briefly describe the type of design flaw affecting the class?*

3. *If yes to question (1), may you please rate the criticality of the design flaw from ① (very low) to ⑤ (very high)?*

We asked the first question to make sure that the contacted developers perceived classes as affected by code smells. If not, they could not obviously

provide meaningful information on their criticality, and the answers were discarded. Otherwise, we further asked to explain the design problem perceived, so that we could understand if developers were actually aware of the specific smell. When receiving the answers, we checked if the explanation given by developers was in line with the definition of the smell: for instance, one developer was contacted to rate the criticality of a *Blob* class and explained that *"[the class] is a well-known problem, it is huge in size and has high coupling"*, thus indicating that s/he correctly recognized the smell we were proposing to him/her. In these cases, we considered the answer to the third question valid, otherwise we discarded it. Note that if a code smell was detected in the same class more than once, we did not send any e-mail to not bother the developers multiple times for the same class.

As an outcome, we sent a total of $1,733$ e-mails to 372 distinct developers, i.e., an average of 0.77 e-mails per month per developer, while 487 code smells affected the same classes multiple times and, therefore, we avoided sending e-mails for them. Moreover, we could not assess the criticality of 310 code smells because the 139 developers responsible for them did not reply to our e-mails. Also, we had to discard 91 answers received since the contacted developers did not perceive the presence of code smells, i.e., they answered *'no'* to question (1).

Hence, we finally gathered $1,332$ valid answers coming from 233 developers: the high response rate (62%) is in line with previous works that implemented a similar recruitment strategy [238, 273] and indicates that contacting developers immediately after their activities with short surveys not only increases the chances of receiving accurate answers [273], but also helps increasing their overall responsiveness. As a final note, the $1,332$ code smell instances evaluated were almost equally distributed among the four considered types of design flaw: indeed, we had answers for 341 *Blob*, 349 *Complex Class*, 313 *Spaghetti Code*, and 329 *Shotgun Surgery* instances. Also, the criticality values assigned by developers to each smell type were almost uniformly distributed over the possible ratings (① to ⑤).

6.2 A NOVEL CODE SMELLS PRIORITIZATION APPROACH AND ITS EVALUATION

The *goal* of our study is to define and assess the feasibility of using a machine learning-based solution to prioritize code smells according to their perceived criticality, with the *purpose* of providing developers with recommendations that are more aligned to the way they actually refactor source code. The *perspective* is of both practitioners and researchers: the former are interested in adopting more practical solutions to prioritize refactoring activities, while the latter are interested in assessing how well machine learning can be employed to model developer's criticality of code smells.

6.2.1 *Research Questions*

The empirical study revolves around three main research questions (**RQ**s). We started with the definition of a machine learning-based approach to model the developer's perceived criticality of code smells. Starting from the dataset built following the strategy reported in Section 6.1, we defined dependent and independent variables of the model as well as the appropriate machine learning algorithms to deal with the problem. These steps led to the definition of our first research question:

RQ$_1$. *Can we predict developer's perception of the criticality of a code smell?*

Besides assessing the model as a whole, we then took a closer look at the contributions given by the independent variables exploited, namely what are the features that help the most when classifying the perceived criticality of code smells. This step allowed us to verify some of the conjectures made when defining the set of independent variables to be used. Hence, we asked:

RQ$_2$**.** *What are the features of the proposed approach that contribute most to its predictive performance?*

As a final step of our investigation, we considered the literature in the field to identify existing code smell prioritization approaches that can be used as baselines, thus allowing us to assess how useful our technique can be when compared to existing approaches. This led to our final research question:

RQ$_3$. *How does our approach perform when compared with existing code smell prioritization techniques?*

In the next sections, we describe the methodological details of the evaluation of the proposed approach.

6.2.2 *RQ$_1$. Defining and assessing the performance of the prioritization approach*

To address **RQ$_1$**, we defined a novel code smell prioritization approach that aims at classifying smell instances based on the developer's perceived criticality using machine learning algorithms. This implied the definition of an appropriate set of independent variables able to predict the dependent variable, i.e., the developer's perception, as well as the proper algorithms and their configuration.

Dependent Variable. As a first step, we defined the developer's perceived criticality of code smells as a variable that the model has to estimate. The dataset described in Section 6.1 reports, for each code smell instance, a value ranging from 1 to 5 describing the perceived criticality of that instance. Thus, we mapped the problem as a classification one [54], namely we took into account the case in which the learner has to classify the criticality of code smells in multiple categorical classes [54]. In this case, we converted the integers of our dataset in nominal values in the set {*low*, *medium*, *high*}: if a code smell instance was associated to 1 or 2 in the original dataset, then we considered its perceived criticality as *low*; if it was equal to 3, we converted its value in *medium*; otherwise, we considered its criticality as *high*. With this mapping, we merged the values assigned by developers in order to build three main classes. This was a conscious decision given by experimental tests: indeed, when experimenting with a 5-point classification problem, we

Table 6.2: Software metrics used as independent variables split by categories - The motivations for their use are also reported.

Metric	Acronym	Description
Product metrics. Cohesion, coupling, and complexity may lower code quality and affect the perceived criticality of code smells [227, 283, 288].		
Lines of Code	LOC	Amount of lines of code of a class excluding white spaces and comments.
Lack of Cohesion of Methods [50]	LCOM5	Number of method pairs in a class having no common attribute references.
Conceptual Cohesion of Classes [183]	C3	Average cosine similarity [16] computed among all method pairs of a class.
Coupling between Object Classes [50]	CBO	Number of classes in the system that call methods or access attributes of a class.
Message Passing Coupling [50]	MPC	Number of method calls made by a class to external classes of the system.
Response for a Class [50]	RFC	Sum of the methods of a class, i.e., number of methods that can potentially be executed in response to a message received by an object of a class.
Weighed Methods per Class [50]	WMC	Sum of the McCabe cyclomatic complexity [191] computed on all methods of a class.
Readability [36]	Read.	Measure of source code readability based on 25 features, e.g., number of parentheses per lines of code.
Process metrics. The amount of activities made on smelly classes may affect the developer's perceived criticality [162, 174, 262].		
Average Commit Size	AVG_CS	Average number of classes that co-changed in commits involving a class.
Number of Changes	NC	Number of commits in the change history of the system involving a class.
Number of Bug Fixes	NF	Number of bug fixing activities performed on a class in the change history of the system.
Number of Committers	NCOM	Number of distinct developers who performed commits on a class in the change history of the system.
Developer-related metrics. Experience and workload of developers may affect the perceived criticality of code smells [30, 43, 44, 304].		
Developer's Experience [30]	EXP	Average number of commits of the committers of a class.
Developer's Scattering Changes [68]	DSC	Average number of distinct subsystems in which the committers of a class made changes.
Development Change Entropy [119]	CE	Shannon's entropy [269] computed on the number of changes of a class in the change history of the system.
Code Ownership [30]	OWN	Number of commits of the major contributor of a class over the total number of commits for that class.
Code smell-related metrics. Persistence of code smells and availability of refactoring opportunities may affect the developer's perception [223, 243].		
Persistence	Pers.	Number of subsequent major/minor releases in which a certain smell affects a class.
Intensity [187]	Sev.	Average distance between the actual metric values used for the detection of code smells and the corresponding thresholds considered by the detectors to distinguish smelly and non-smelly elements.
Refactorable	Ref.	Existence of refactoring opportunities for a class, as detected by automated tools.
Number of Refactoring Actions	NR	Number of previous refactoring operations made by developers on a class.

observed that several misclassifications were due to the approach not able to correctly distinguish (i) *very-low* from *low* and (ii) *high* from *very-high*. Thus, we opted for a 3-point classification.

Independent Variables. To predict the perceived criticality of code smells, we considered a set of features able to capture the characteristics of classes under different angles. Table 6.2 summarizes the families of metrics considered, the rationale behind their use, and the specific indicators measured.

Previous research has not only shown that product metrics can indicate actual design problems in source code [42, 207], but also lead developers to recognize the presence of sub-optimal implementation solutions that would deserve some refactoring [227, 283, 288]. For these reasons, we considered four types of product metrics, such as (i) size, (ii) cohesion, (iii) coupling, and (iv) complexity metrics. For each of these types, we selected indicators having different nature (e.g., structural aspects of source code rather than textual components) and able to capture in different ways the considered phenomena

(e.g., we computed source code complexity using both the McCabe metric and readability, which targets a more cognitive dimension of complexity).

While product metrics can provide indications about the structure of source code, we complemented them with orthogonal metrics that capture the way the code has been modified as well as who was responsible for that, i.e., process and developer-related metrics. Indeed, the developer's perception of criticality may be not always due to the complex structure of source code, but rather to the problems it causes during the evolution process [236, 262]; similarly, the criticality of code smells may be perceived differently depending on whether the maintainer is an expert of the class or not [30, 44]. Hence, we selected a number of metrics related to those aspects: for instance, we computed the average number of co-changing classes for the smelly class (AVG_CS) to assess whether smells that often change with several classes are perceived as more critical by developers or the number of previous bug fixing involving the smelly class to observe if classes that are more fault-prone are actually those perceived more critical. We also computed measures of experience and ownership of developers working on the smelly class to test whether these factors influence the developer's ability to work on it.

Finally, we took into account some specific metrics related to code smells: the idea here is that a number of aspects connected to the smell itself may be relevant for developers when assessing its criticality. In particular, the continuous presence of smell over the history of the project (i.e., Pers.) may influence the ability of developers to recognize its harmfulness better. Much in the same way, the presence of refactoring opportunities (Ref.) or even the number of previous refactoring actions done on the smelly class (NR) may affect the perception of developers. Finally, we also considered the code smell intensity, which is a measurable amount of intensity of a certain code smell instance [187]: we included this metric to understand whether there is a match between an "objective" measurement of code smell severity and its real perceived criticality.

From a technical perspective, we employed the tool by Spinellis [282] to compute product metrics, the one made available by Buse and Weimer [36] for the computation of the readability index, and PyDriller [279] to compute

process and developer-related metrics. As for the smell-related indicators, we developed our own tool to compute Pers. and NR. Starting from the release R_i of the projects taken into account, the former metric counts in how many consecutive previous major and minor releases—identified using the corresponding GIT tags—the considered smell was present, according to the employed detectors. The latter metric, instead, was computed by mining the messages of commits involving the smelly classes and looking for the presence of keywords recommended in [313], e.g., *'refactor'* or *'restructure'*. The intensity of code smells has been assessed using the tool by Marinescu [187], which computes the average distance between the actual code metric values of the smell instance and the thresholds fixed by the detection rules. Finally, the Ref. metric was computed by (i) running JDEODORANT [81], an existing refactoring recommender that covers all refactoring actions associated to the considered code smells, and (ii) putting the metric to 1 if the tool retrieved at least one recommendation, 0 otherwise.

Machine Learning Algorithms. Once we had computed dependent and independent variables, we proceeded with the definition of the machine learning algorithms to be used.

In order to perform the classification, we investigated the use of multiple algorithms [54], i.e., Random Forest, Logistic Regression, Vector Space Machine, Naive-Bayes, and Multilayer Perceptron, in order to assess what is the one giving the best performance. Note that, before running the algorithms, we first applied a forward selection of independent variables using the CORRELATION-BASED FEATURE SELECTION (CFS) approach [169], thus mitigating possible problems due to multi-collinearity of features [215]. Then, we configured classifiers' hyper-parameters by exploiting the GRID SEARCH [29] algorithm.

Training/Testing the Model. We built different models for each code smell considered in the study, so the training data is represented by the set of observations available for a certain smell in the collected dataset: the distribution is almost uniform for all the criticality values. This aspect affected our decision to not apply any balancing algorithm. On the one hand, there are no classes requiring to be balanced with respect to the others. On the other hand,

previous findings have shown that balancing code smell-related datasets can even damage the performance of the resulting models.

To train the model, we employed a 10-fold cross-validation strategy [147]. The process is repeated ten times so that each fold will be the test set exactly once.

Performance Assessment. We evaluated the performance of the experimented model by analyzing confusion matrices, obtained from the testing strategy described above, reporting the number of true and false positives as well as the number of true and false negatives. We analyzed these matrices by first computing precision, recall, F-Measure, and the Matthew's Correlation Coefficient (MCC) [16].

6.2.3 RQ_2. *Explaining the Proposed Approach*

In the context of $\mathbf{RQ_2}$, we took a deeper look into the performance of the best model coming from the previous research question. We aimed at understanding the value of the individual metrics selected as independent variables; this step could possibly help us explaining why the proposed approach works (or not) when predicting the criticality of code smells. To this aim, we employed an *information gain* algorithm [257].

To analyze the resulting rank and have statistically significant conclusions, we finally exploited the Scott-Knott Effect Size Difference (ESD) test [295]. This is an effect-size aware variation to the original Scott-Knott test [268] that has been recommended for software engineering research in previous studies [129, 160, 294] as it (i) uses hierarchical cluster analysis to partition the set of treatment means into statistically distinct groups according to their influence, (ii) corrects the non-normal distribution of an input dataset, and (iii) merges any two statistically distinct groups that have a negligible effect size into one group to avoid the generation of trivial groups. As effect size measure, the test relies on Cliff's Delta (or d) [108]. To compute the test, we used the publicly available implementation[5] provided by Tantithamthavorn *et al.*[295].

5 Link: https://github.com/klainfo/ScottKnottESD

6.2.4 RQ_3. *Comparison with the state of the art*

Finally, we investigated whether and to what extent the proposed code smell prioritization approach overcomes the performance of existing techniques. This step is paramount to understand the novelty of our solution and how it may support developers better than the baseline approaches.

The two closest techniques with respect to the one proposed herein are those by Vidal *et al.*[317] and Arcelli Fontana and Zanoni [86]: we set them as initial baselines for the comparison. In the former, the authors proposed SPIRIT, a semi-automated technique that relies on three main criteria, namely (1) *stability*, i.e., number of previous changes applied on a smelly class over the number of total changes applied on the system, (2) *relevance*, i.e., the relative importance of the class within the system according to the feedback given by a developer, and (3) *modifiability scenarios*, i.e., the number of possible use cases of the application that risk to be impacted by the presence of the smell according to the opinion of an expert. The three criteria are then combined through a weighted average, where the weights are assigned by the user of the tool. As the reader might have noticed, SPIRIT explicitly requires the intervention of an expert to be employed in practice: indeed, the technique has been tested in an industrial case study involving a JAVA project affected by a total of 47 code smells and requiring the interaction of core developer of the subject application. For this reason, we could not use it as a baseline for a *in-vitro* assessment of our proposed approach and we plan to perform a comparison with SPIRIT in our future research agenda.

As for the technique proposed by Arcelli Fontana and Zanoni [86], this is a machine learning-based solution that relies on 61 product metrics to predict how critical a certain code smell instance is. This technique can be fully automated and, therefore, we could use it as the baseline for our study and run it using the same dependent variable, training, validation strategy, and dataset employed to validate our approach. Once obtained the output from the baseline, we compared it with ours by means of the same set of metrics used in $\mathbf{RQ}_1$, i.e., precision, recall, F-Measure, MCC, and AUC-ROC.

Table 6.3: **RQ_1 - RQ_3.** Confusion matrices obtained when running the proposed model against our dataset.

Model	Class/Smell	Blob			Complex Class			Spaghetti Code			Shotgun Surgery		
		Non-severe	Medium	Severe	Non-severe	Medium	Severe	Non-severe	Medium	Severe	Non-severe	Medium	Severe
Our approach	Non-severe	54	10	6	46	17	27	57	11	2	46	37	13
	Medium	6	171	18	14	176	0	5	151	6	8	134	10
	Severe	1	10	65	6	1	62	0	7	74	5	16	60
Baseline	Non-severe	16	40	14	45	43	2	37	19	21	11	83	6
	Medium	12	174	9	21	161	8	0	172	0	18	122	12
	Severe	8	23	45	7	48	14	13	0	79	6	79	4

Table 6.4: **RQ_1 - RQ_3.** Weighted Average of the performance achieved by the experimented models against our dataset.

Code smell	Model	Prec.	Rec.	F-Meas.	MCC	AUC-ROC
Blob	Our approach	86%	85%	85%	75%	89%
	Baseline	66%	69%	66%	44%	78%
Complex Class	Our approach	79%	81%	80%	71%	89%
	Baseline	62%	63%	63%	33%	76%
Spaghetti Code	Our approach	90%	88%	89%	83%	92%
	Baseline	83%	85%	84%	77%	89%
Shotgun Surgery	Our approach	74%	71%	72%	61%	78%
	Baseline	33%	40%	35%	32%	61%

6.3 ANALYSIS OF THE RESULTS

This section reports the results of our study, presenting each research question independently.

6.3.1 *RQ_1. The Performance of our Model*

In the context of **RQ_1**, we aimed at assessing how well can we predict the perceived criticality of code smells. Table 6.3 reports the confusion matrices obtained when running the proposed approach against our dataset of four code smell types, while Table 6.4 presents the weighted average performance for each code smell. For the sake of space limitations, we only report the results achieved with the best classifier, i.e., Random Forest.

In the first place, it is worth noting that the performance values of our model are rather high and, indeed, it has an F-Measure that ranges between 72% and 85%. This indicates that, in most of the cases, our model can accurately

classify the severity perceived by developers. The worst case is represented by *Shotgun Surgery*, where the model has an F-Measure of 72% and an AUC-ROC of 61%. On the one hand, the former metric still indicates that the model is able to correctly classify most of the instances of our dataset. On the other hand, the latter suggests that the ability of separating criticality classes may be further improved; this is also visible when considering the confusion matrix for this smell (Table 6.3), where we noticed that in 52% of cases the model classified non-severe code smells as medium or severe cases. An example is represented by the class `security.JackrabbitAccessControlManager` of the JACKRABBIT project. This class has 164 lines of code and has been detected as smelly because every time it is changed an average of other 11 classes are also modified. Nevertheless, it has been subject to a relatively low number of changes (19) and defects (1), likely being less harmful than other instances of the smell. Analyzing the other misclassified cases, we noticed a similar trend: the model tends to misclassify instances because it is not always able to learn how to balance the information coming from the number of classes to be modified with the smelly one and the actual number of changes that involve the smelly instance. As such, we can claim that possible improvements to the classification model may concern the addition of combined metrics, e.g., the ratio between number of co-changing classes and number of previous changes of the smelly class.

As for the other code smells considered, the performance values are higher and all above 80% and 70% in terms of F-Measure and AUC-ROC. Hence, we can claim that the proposed model can be effectively adopted by developers to prioritize code smell instances. The best result is the one of *Spaghetti Code* (F-Measure=89%, AUC-ROC=92%): in this case, the model misclassifies only 31 cases (10% of the instances). By looking deeper at those cases, we could not find any evident property of the source code leading to those false positives. A similar discussion can be drawn when considering the *Blob* and *Complex Class* code smells. Part of our future research agenda includes the adoption of mechanisms able to better describe the functioning of the learners used for the classification, e.g., explainable AI algorithms [110].

Summing Up: The proposed model has an F-Measure ranging between 72% and 85%, hence being accurate in the classification of the perceived criticality of code smells. The worst case relates to Shotgun Surgery, where the model misclassifies non-severe instances because of its partial inability to take into account other process-related information like number of changes involving the smelly classes.

Table 6.5: **RQ_2.** Information Gain of the independent variables of our approach. For space limits, only metrics providing significant contributions are reported.

Code smell	Metric	Mean	SK-ESD
Blob	RFC	0.65	68
	LCOM5	0.57	66
	NF	0.56	66
	DSC	0.55	64
	CBO	0.45	64
	WMC	0.42	64
	C3	0.35	45
	LOC	0.34	41
Complex Class	CBO	0.59	71
	WMC	0.54	69
	LCOM5	0.54	69
	Read.	0.54	69
	NC	0.50	54
	DSC.	0.49	51
	EXP	0.27	33
	RFC.	0.25	31
Spaghetti Code	Read.	0.65	53
	NF	0.57	46
	C3	0.38	41
Shotgun Surgery	NC	0.32	44
	LCOM5	0.31	39
	AVG_CS	0.24	33
	Pers.	0.17	21

6.3.2 *RQ_2. Features Contributing to the Model*

Table 6.5 reports the list of features contributing the most to the performance of the proposed model. As shown, each code smell has its own peculiarities.

To classify *Blob* instances, the model mostly relies on structural metrics that capture complexity (RFC, WMC), cohesion (LCOM5, C3), and coupling (CBO) of the source code: basically, it means that the criticality of this code smell is given by a mix of various structural factors and cannot be described by just looking at them independently. At the same time, the number of previous defects affecting those instances (NF) as well as the workload of the committers (DSC) have a non-negligible effect. As such, on the one hand we can confirm previous findings that showed historical and socio-technical factors as relevant to manage code smells [228, 238]. On the other hand, our findings suggest that these metrics may possibly be useful for detecting code smells in the first place or even filtering the results of currently available detectors, so that they may give recommendations that are closer to the developer's perceived criticality. Finally, the lines of code also contributes to the model, being however not the strongest factor—confirming again previous findings in the field [228, 234].

When considering *Complex Class*, a similar discussion can be done. While the most impactful metrics concern with the structure of the code (CBO, WMC, LCOM5), other metrics seem to have a relevant effect on the classification model. In particular, readability is the strongest factor after code metrics, indicating that developers consider comprehensibility important when prioritizing this code smell. Other relevant factors are the number of previous changes of classes (NC) and socio-technical aspects like experience and workload of the committers (EXP, DSC): again, this result confirms that the management of code smells may require additional information than structural aspects of source code [304].

Surprisingly, when considering *Spaghetti Code* instances we noticed that no structural factors strongly influence the classification. Readability is indeed the key factor leading developers to prioritize instances of this smell, followed by the number of previous defects affecting those classes (NF) and by the conceptual cohesion of classes (C3). Hence, it seems that developers prioritize instances of this smell that are semantically incoherent or that suffered from defects in the past. Our findings could again be used by code smell detection

and filtering approaches to tune the list of recommendations to provide to developers.

Finally, the prioritization of the *Shotgun Surgery* smell is mainly driven by process-related factors. Not only the number of changes (NC) is the most powerful metric, but also the number of co-changing classes (AVG_CS) turned out to be relevant. Also, this is the only case in which the persistence of the smell (Pers.) appeared to impact the classification. These result seem to confirm that developers assess the severity of this code smell based on the intensity of the problem [227, 288], i.e., when number of changes or co-changing classes is high or when the problem is constantly affecting the codebase. Furthermore, the cohesion of the class (LCOM5) affects the classification, even though at a lower extent if compared to the contribution given for other code smell types.

Summing Up: The developer's perceived criticality of code smells represents a multi-faceted problem that can be tackled considering a mix of metrics having different nature (e.g., structural or historical) and working at various levels of granularity (e.g., process or socio-technical aspects).

6.3.3 RQ_3. *Comparison with the state of the art*

We compared the proposed model with a baseline. The results are reported in Tables 6.3 and 6.4, where we show confusion matrices and weighted performance values obtained when running the baseline against our dataset, respectively. Also in this case, we report the results obtained with the best classifier, that in this case was Logistic Regression—confirming the findings of the original authors [86].

In the first place, we can notice that the baseline is decently accurate and, indeed, its F-Measure values on *Blob*, *Complex Class*, and *Spaghetti Code* range between 63% and 84%. The exception is *Shotgun Surgery* (F-Measure=35%), where the baseline fails the classification in most of the cases. Despite its performance, however, the baseline never outperforms our technique. While this is especially true when considering *Shotgun Surgery*

(-37% of F-Measure, -17% of AUC-ROC), also for the other code smells the difference is non-negligible: the F-Measure is 19%, 17%, and 5% lower than our model for *Blob*, *Complex Class*, and *Spaghetti Code*, respectively.

The main reason for these differences is likely imputable to the metrics employed. As shown in $\mathbf{RQ}_2$, structural aspects of source code can only partially contribute to the classification of the developer's perceived criticality of code smells and, as such, the inclusion of factors covering other dimensions better fits the problem.

Of particular interest is the analysis of the results for the *Spaghetti Code* smell, where the baseline has the highest performance despite the fact that our findings in $\mathbf{RQ}_2$ reported structural aspects to be negligible. The baseline employs a variety of metrics that can capture different aspects of source code (e.g., coupling or cohesion) under different angles (e.g., by considering the lines of code with and without access methods). Some of the complexity metrics are highly correlated to readability of source code and its fault-proneness and, as such, they have the effect of "simulating" the presence of metrics like the one found to be relevant in $\mathbf{RQ}_2$. This claim is supported by an additional analysis in which we compute the correlation (using the Spearman's test) between the metrics used by the baseline and those which turned out to be relevant in our previous analysis (Read., NF, and C3): we discovered that five of them (i.e., WMCNAMM_type, NOMNAMM_type, AMW_type, CFNAMM_type, and num_final_static_attributes) are highly correlated, i.e., ρ>0.7, to at least one of the variables found in $\mathbf{RQ}_2$.

In conclusion, based on our findings we can claim that an approach solely based on structural metrics cannot be as accurate in the classification of the perceived criticality of code smells as a technique that includes information coming from other sources, confirming again that the problem of code smell management should be tackled in a more comprehensive manner.

Summing Up: The proposed model is, on average, 20% more accurate than the baseline when classifying the perceived criticality of code smells. Only in the case of Spaghetti Code the usage of multiple structural metrics can lead to results similar to those of our model.

6.4 CONCLUSION

This chapter presented a novel code smell prioritization approach based on the developers' perceived criticality of code smells. We exploited several aspects related to code quality to predict the criticality of code smells, computed by collecting feedback from original developers about their perception of 1,332 code smell instances. Then, we applied several machine learning techniques to classify the code smell criticality in a three-level variable, and compared their results with a state-of-the-art tool. The results reported Random Forest to be the best machine learning algorithm with an F-measure ranging between 72% and 85%. Moreover, we found that our approach is, on average, 20% more accurate than the considered baseline when classifying the perceived criticality of code smells.

7

THREATS TO VALIDITY, DISCUSSION, AND IMPLICATIONS

In this chapter we report some aspects that might have threaten the validity of the results achieved in our empirical studies and in-depth discuss our main findings to answer our first two high-level research questions.

7.1 THREATS TO VALIDITY

This section discusses the main threats to validity and explains how we mitigated them, following the guidelines provided by Wohlin [327].

7.1.1 *Threats to Construct Validity*

Threats in this category are related to the relation between theory and observation. In our studies, a threat might be represented by the datasets used for our empirical investigations. As for the first four studies (Chapter 3 to Chapter 5) we considered several factors for the dataset selection such as heterogeneity or the presence of manually-validated data, however we have to consider that they may contain possible discrepancies or inaccuracies, such as labeling errors or some positive instances that might have been overlooked. For instance, some of our datasets contain multiple releases from the same project. In such a context, it could happen that different systems have different weights in the dataset: systems having more release have an higher weight, therefore a higher impact on the overall performance. However, to mitigate this aspect and, at the same time, avoid possible effects due to dependencies among successive releases, we evaluated the models performance relying on aggregated assessment metrics to have a clearer overview of the performance [12].

In our last study, Chapter 6, we needed to collect the developer's perceived criticality of a set of code smells. To this aim, we followed a similar strategy as previous work [238, 273]: we monitored nine large open-source systems for 6 months and inquired the original developers as soon as they modified smelly classes in order to let them rank how harmful the involved code smells actually were. In so doing, we adopted some precautions. Firstly, we detected code smells using state-of-the-art tools [197, 228] that showed high accuracy, yet checking their output to remove false positives; in any case, we cannot exclude the presence of false negatives since these detectors have been validated on different datasets. Secondly, we asked preliminary questions on whether they perceived the presence of a design issue in the proposed class and recognized the same problem they were contacted for. These questions aimed at ensuring that developers were really aware of the code smells they were assessing and, thus, could provide us with reliable feedback. Of course, we are aware that some of the developers might be peripheral contributors without the experience required to assess the harmfulness of code smells. To account for this aspect, we conducted a follow-up verification of the role of the subject developers within their corresponding projects: to this aim, we computed the number of commits they performed (i) over the entire change history of their projects and (ii) on the specific classes they were contacted for. As a result, we discovered that all our respondents have contributions that exceed the median number of commits made by all project's developers both in terms of changes done over the history and on the smelly classes objects of our inquiry. In conclusion, we can argue that the dataset collection method is sound and allows a reliable analysis of the perceived code smells criticality. Furthermore, the perceived criticality assigned by developers when building the dataset might have been influenced by the co-occurrence of multiple code smells [1]. We mitigated this problem by presenting to developers classes affected by single code smell types among those considered in this chapter, e.g., we only presented cases where a *Blob* did not occur with any of the other smells considered in the study. Nevertheless, we cannot exclude the presence of further design issues among those that we did not consider in the

study. As such, a larger experimentation would be desirable to corroborate our observations.

Another common threat is related to the construction of the machine learning models, for which we took several aspects into account that could have possibly influenced the study, i.e., which features to consider, how to train the classifier, etc.. However, we believe that the procedures followed in this respect are precise enough to ensure the validity of the study.

Finally, it is worth remarking that most of the independent metrics computed as well as the algorithms exploited (e.g., the machine learners) were computed by relying on well-tested, publicly available tools. This allowed us to reduce biases due to re-implementation. Their selection was based on convenience, and particularly on the skills that the authors have with them.

7.1.2 *Threats to External Validity*

With respect to the generalizability of our findings, we considered large datasets consisting of systems belonging to different application domains and having different characteristics. Another threat concerns the choice of the machine learning techniques. To mitigate this threat, over the different studies we always compared the top most commonly used classifiers in this field [15]. Finally, also the selection of the code smell to analyze could represent a threat to the external validity. We selected several code smell types that represent a large variety of design issues (e.g., smells related to complexity or excessive coupling between objects). This allowed us to better understand the potential of machine learning techniques for code smell detection as well as their limitations with respect to heuristic-based approaches. Of course, further experiments performed on different datasets and techniques would be desirable and already part of our future research agenda.

7.1.3 *Threats to Conclusion Validity*

As for concerns with the relationship between treatment and outcome, we exploited a set of widely-used metrics to evaluate the experimented techniques

(e.g., precision, recall, MCC) and provided qualitative examples aimed at showing the differences between the compared approaches. Furthermore, we used appropriate statistical tests (e.g., Wilcoxon, Cliff's delta) to support our findings. As for the machine learning model, a possible bias related to the interpretation of the results might have been due to the usage of the 10-fold cross validation. This strategy randomly partitions the set of data to create training and test sets: such randomness might have possibly led to the creation of biased training/test sets that have the consequence of under- or over-estimate the model performance. To account for this aspect, we performed additional analyses: as suggested by Hall *et al.*[115], we ran the experimented model multiple times to assess how stable it is depending on the random splits performed by the validation strategy. Thus, we ran 10 times 10-fold cross validations and, then, we measured the variability of the predictions performed by the model; as a result, we observed that in the great majority of the cases (more than 95%) the predictions do not change over different runs. As such, we can conclude that the results achieved are not influenced by the randomness of the validation strategy.

7.1.4 *Threats to Internal Validity*

These threats are related to the internal factors of the study that might have affected the results. The results we discussed are characterised by a great variability with respect to the smell under analysis. A possible reason could be the metric selection for code smell detection. Indeed, some of the selected metrics could represent a confounding factor threatening the internal validity of the study. To mitigate this threat, we relied on previously defined and validated metrics.

7.2 DISCUSSION AND IMPLICATIONS

This section provides the answers to our first two high-level research questions (i.e., RQ_a and RQ_b) through a deep discussion of the results achieved in the studies presented in Part I.

Table 7.1: Type I and Type II Errors Achieved in the Overall Evaluation

	Naive Bayes		Optimistic Constant		Pessimistic Constant		Random	
	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II
God Class	1,263 (0.90%)	65 (0.10%)	144,798 (99.60%)	0 (0.00%)	0 (0.00%)	509 (0.40%)	72,683 (50.00%)	251 (0.20%)
Spaghetti Code	2,269 (1.40%)	1,009 (0.60%)	159,436 (99.10%)	0 (0.00%)	0 (0.00%)	1,443 (0.90%)	79,669 (49.50%)	690 (0.40%)
Class Data Should Be Private	874 (0.60%)	770 (0.50%)	142,558 (99.20%)	0 (0.00%)	0 (0.00%)	1,150 (0.80%)	71,221 (49.50%)	589 (0.40%)
Complex Class	1,303 (1.00%)	282 (0.20%)	127,538 (99.50%)	0 (0.00%)	0 (0.00%)	669 (0.50%)	63,507 (49.50%)	335 (0.30%)
Long Method	15,449 (1.20%)	2,101 (0.20%)	1,283,312 (99.60%)	0 (0.00%)	0 (0.00%)	4,763 (0.40%)	641,914 (49.80%)	2,431 (0.20%)

7.2.1 RQ_a - *The capabilities of machine learning-based algorithms for code smell detection*

In our first high-level research question ($\mathbf{RQ}_a$) we wondered about the capabilities of machine learning-based algorithms for code smell detection. The results of the study presented in Chapter 3 provided a number of insights to answer RQ_a that deserve some further considerations.

On the Performance of Machine Learning Models As we have observed in Section 3.2, the performance of machine learning techniques are not as good as the one of heuristic approaches. To further investigate the potential of these techniques, we performed an additional analysis aimed at comparing the model with three simple baselines such as: (i) the OPTIMISTIC CONSTANT classifier, that always classifies an instance as smelly; (ii) the PESSIMISTIC CONSTANT classifier, that always classifies an instance as non-smelly; and (iii) a RANDOM classifier, which randomly classifies an instance as smelly or non-smelly. Should the performance of the model be lower than any of this baseline, it would indicate a major threat to the usability of the model in practice. As previously done in literature [111], we performed this comparison in terms of *Type I* and *Type II* errors, i.e., computed as the total number of false positive and false negative errors.

Table 7.1 reports the results achieved. We can observe that, for each of the classifiers, the total number of errors (i.e., Type I + Type II) is independent from the smell to detect. The total number of errors in percentage is between 1% and 2% for NAIVE BAYES, higher than 99% for OPTIMISTIC CONSTANT, less than 1% for PESSIMISTIC CONSTANT and around 50% for RANDOM CLASSIFIER. This means that the PESSIMISTIC CONSTANT outperforms all the other classifiers producing a lower number of errors.

Of course, this result was due again to the unbalanced nature of the problem. However, this has a key implication for the research community: based on our results, *machine learning seems still unsuitable for code smell detection* which also involves practitioners who currently cannot use this approach in practice. The inclusion of orthogonal metrics as independent variables (e.g., process indicators), the adoption of ensemble techniques [69], the experimentation of different training strategies (e.g., cross-project models) are just some of the research fields that would require further attention in the future.

On the Performance of Heuristic Approaches One of the most surprising result of our study in Chapter 3 concerns the fairly low performance achieved by DECOR over the considered dataset. Specifically, while the recall of the approach was in line with the one stated in literature [197], we found its precision to be extremely low. We see two main motivations behind this result. First, in our study we employed DECOR with a larger variety of code smells with respect to previous work [139, 228, 234]: therefore, we tested its performance in the wild, showing some limitations of the technique when employed for the detection of certain code smell types. Secondly, we relied on a manually-validated dataset containing real instances: as shown by previous work [71], the composition of the dataset might influence the performance of a technique; this is especially true in the case of code smell detection, where a detector should recognize code smells over datasets that are both unbalanced (i.e., limited number of actual instances) and noisy (i.e., the presence of several smell types might interfere and make the detection rules less effective).

Thus, while heuristic techniques still slightly outperform machine learning models, *the problem of detecting code smells using heuristics is still far from being solved*. We believe that our findings support the preliminary research efforts conducted to filter code smell candidates output by the detectors to reduce the false positive rate, thus improving their precision [84]. At the same time, we also envision further research on how to limit the interaction of multiple code smells with similar characteristics on the performance of code smell detectors: to this aim, we envision the concept of *local smell detection*, that, similarly to what has been done in defect prediction [193], would have

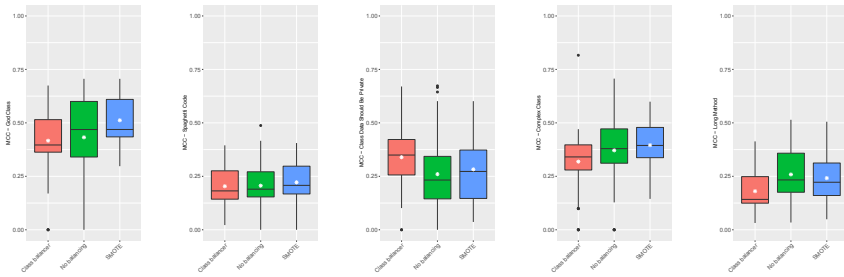

Figure 7.1: Boxplots representing the MCC values obtained by NAIVE BAYESIAN trained applying different balancing strategies for all the considered code smells

the goal of clustering similar classes first (possibly positively affecting the interaction problem) and then apply the detector that is most suitable for the classes of a cluster.

7.2.2 *RQ_b - The limitations of machine learning-based algorithms for code smell detection*

While assessing the performance of machine learning-based approaches for code smell detection in order to answer RQ_a, we found some major limitations of these approaches. This section provides discussions about such limitations and possible solutions to overcome them.

On the Role of Data Balancing In the study presented in Chapter 3, we consciously chose to not use balancing techniques when building our code smell detection model. This choice was due to experimental tests where we compared the balanced version of the model with the non-balanced one. As balancing algorithm, we applied the SYNTHETIC MINORITY OVER-SAMPLING TECHNIQUE, (SMOTE) [48].

The results of our analyses are shown in Figure 7.1. At a first sight, it may seem that the balanced model achieves better performance for most of the considered smells (except the one created for *Long Method*). However, by further investigating this result we realized that such performance is biased by the high number of cases in which the data balancing algorithm failed,

not producing any valid predictions. In particular, the problem of code smell detection is strongly unbalanced and, in many cases, SMOTE does not have a feature space large enough to perform a balancing, thus producing a failure. Such failures do not contribute to the computation of the evaluation metrics used for performance assessment. As such, the interpretation of the results would have been biased by the absence of many predictions.

To further investigate this aspect, we tried to reduce the failure rate by considering the default CLASS BALANCER provided by WEKA: differently from SMOTE, it performs a very simple balancing that has the goal of re-weighting the instances in each class to obtain the same total class weight. Thus, it theoretically reduces the number of failures as it does not require a large feature space. As expected, the failure rate was actually reduced but, however, the results (shown in Figure 7.1) showed lower performance than the ones produced by the non-balanced model in most cases.

In Chapter 4 we conducted a deeper analysis on the role of data balancing. However, since we described the results only in quantitative terms, herein we provide a deeper discussion in qualitative terms. Specifically, we analyse the overlap between the results achieved by the models using different balancing technique to understand which instances they predict and whether these are complementary.

Figure 7.2 shows the misclassified instances obtained by the five models using the data balancing techniques and the *No-balancing* model. The axes represent the features of the model: the *x-axis* is $ELOC$, while *y-axis* is $NMNOPARAM$. '+' data points represent false negatives, while 'x' data points represent false positives. We describe this case because it nicely explains the behaviours of the balancing techniques. The model is built with two features, thus making it easier to analyse than models trained with many more features.

The results confirm what we previously reported. In particular, *One-Class Classifier* exhibited a high level of recall but a very low precision. This means that for code smell detection training only on the instances belonging to the minority class is not effective because these few instances poorly represent the smelly classes. A similar result is obtained for *Cost-Sensitive Classifier*

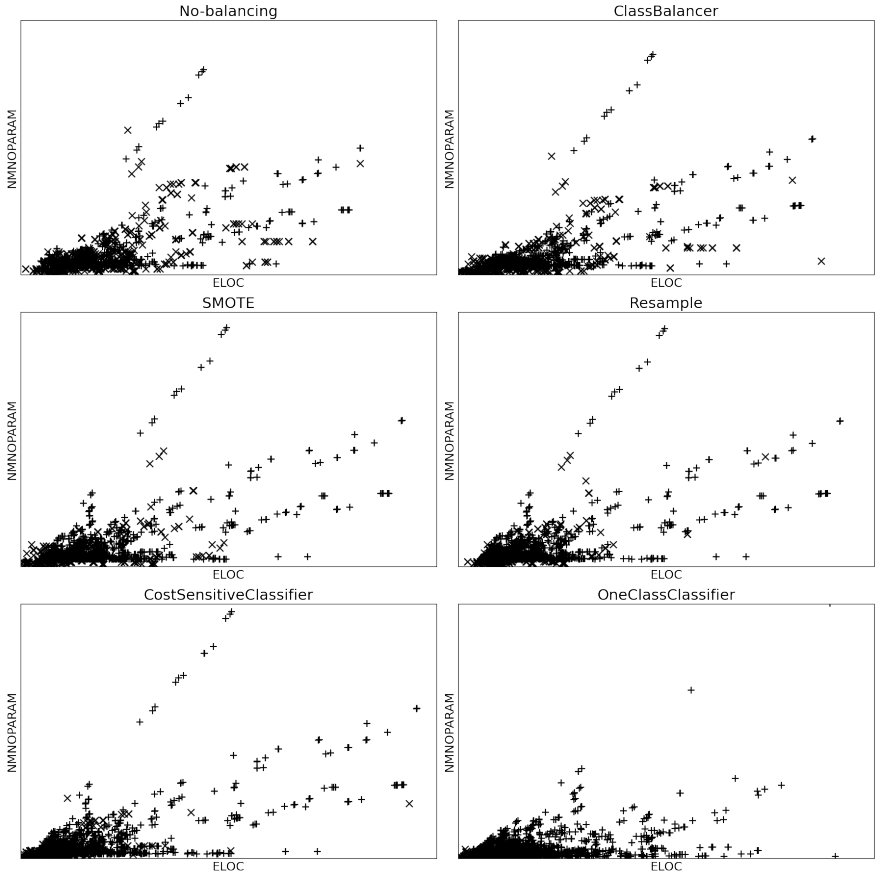

Figure 7.2: Scatterplots representing the misclassified instances obtained by NAIVE BAYESIAN trained applying different balancing strategies for *Spaghetti Code*. '+' data points are false negatives, while 'x' are false positives.

which had poor precision. In particular, we notice that many points were misclassified as true, even if they were false (i.e., false negative). We can argue that giving higher weights to the instances belonging to the minority class is not effective. When analysing at the *Oversampling* and *Undersampling*, we observe that their accuracy is similar to that obtained by *No-balancing*. Therefore, we deem that these techniques are ineffective but do not worsen the accuracy achieved by the model trained without balancing. Finally, we note that *SMOTE* can slightly improve accuracy. However, it is worth remarking

Table 7.2: Number of software systems not exhibiting instances of each smell (i.e., No Smells), along with the instances on which SMOTE could not be executed because of lacking instances for that specific smell (i.e., SMOTE Failures).

Code Smell	No Smells		SMOTE Failures		Number of Systems
	#	%	#	%	
God Class	20	16	46	37	125
Spaghetti Code	7	6	16	13	125
Class Data Should Be Private	10	8	11	9	125
Complex Class	30	24	52	42	125
Long Method	68	54	89	71	125
Feature Envy	82	66	96	77	125
Inappropriate Intimacy	3	2	72	58	125
Middle Man	64	51	104	83	125
Refused Bequest	34	27	46	37	125
Speculative Generality	3	2	11	9	125
Long Parameter List	54	22	62	50	125
Brain Repository	90	75	95	79	120
Fat Repository	87	72	94	78	120
Promiscuous Controller	44	37	73	61	120
Brain Controller	52	43	75	62	120

that some balancing techniques can fail to balance the dataset when the number of smelly instances is minimal. We tuned *SMOTE* to rely on the minimum number of smelly neighbour instances (i.e., two). If these are not available, then the algorithm fails, representing a clear disadvantage with respect to the other techniques.

Table 7.2 reports the number and the percentage of failures for each of the code smells under analysis. While for some code smells, there is a minimal number of failures (e.g., *Speculative Generality*), there are also smells in which the analysis fails in the majority of cases. As an example, let us consider the case of *Fat Repository*. This is one of the less frequent code smells, as also reported in Table 4.6: indeed, in 72% of cases, all data balancing fails due to the total absence of smelly instances in the considered system. As for

SMOTE, it fails in 78% of cases (i.e., 72% with no smelly instances and 6% with not enough neighbours).

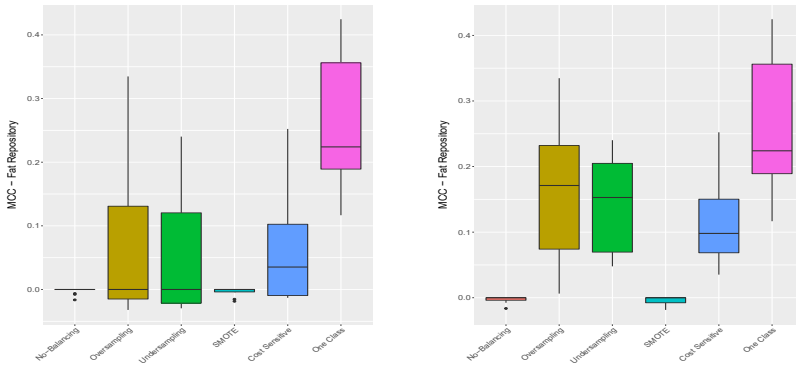

Figure 7.3: Boxplots representing the MCC values obtained by NAIVE BAYESIAN trained with different balancing strategies for the detection of *Fat Repository* code smell. The picture on the left includes the cases in which not all the algorithms could be executed. These cases are filtered out in the picture on the right.

Due to these failures, our analyses have been performed on a smaller population for some code smells. To avoid threatening the significance of results, we conducted a further evaluation in which we included all the systems, regardless of the failures. Figure 7.3 reports an example for *Fat Repository*.The boxplot including all cases is reported on the left of the figure, while on the right side is reported the one excluding failures. Generally, there are small differences in terms of accuracy. Indeed, for both cases, *One Class Classifier* is the most effective data balancing technique.

Overall the results obtained on the different models show that there are no sensible differences in applying or not balancing techniques. This result suggests that tuning data balancing techniques could not be an adequate solution for code smell detection with respect to what achieved in other contexts such as defect prediction [3]. This aspect raises several issues about the feasibility of current machine learning-based approaches. We deem that the meagre number of instances from the minority class (i.e., smelly instances) is the cause of this low effectiveness.

To sum up, data balancing does not significantly improve the effectiveness of machine learning models for code smell detection. Training only on the instances belonging to the minority class or giving them more weight (i.e., as done by *One-Class Classifier* and *Cost-Sensitive Classifier*) is not effective because these few instances poorly represent the minority class. Resampling techniques such as *Oversampling* and *Undersampling* are ineffective but do not worsen the accuracy achieved by the model trained without balancing. Finally, *SMOTE* slightly improves the results, but in case of extremely imbalanced datasets, the training phase fails.

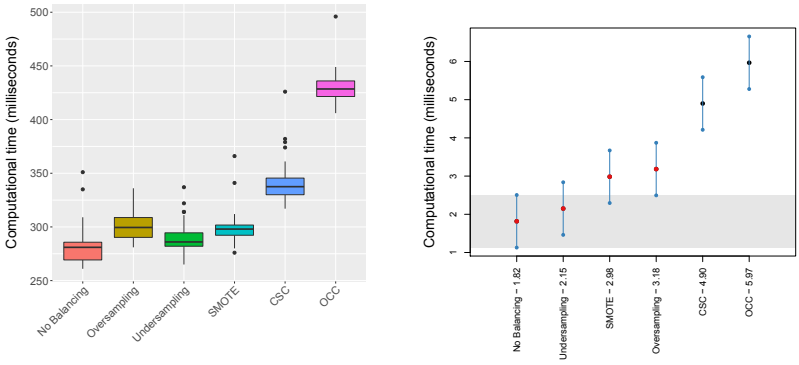

Figure 7.4: Boxplots and Nemenyi results representing the execution time of the different data balancing techniques.

Furthermore, given that data balancing is an additional pre-processing step in ML classification, we also conducted a further analysis to investigate the overhead in terms of time consumption to apply this step. Specifically, we compared the training time of the models configured with different balancing techniques. We performed 30 independent runs training the models on the most extensive system in our dataset, i.e., ECLIPSE 5.2.1. The selection is motivated by a twofold reason. On the one hand, having a higher number of instances should avoid (or at least reduce) failures. On the other hand, a higher number of instances led to longer execution time for all the techniques, and this may allow us to study the overhead better. We considered only one

code smell (i.e., God Class). However, we believe that the results for the other smells should not be very different.

The results in Figure 7.4 highlight that *One-Class Classifier* and *Cost-Sensitive Classifier* take much more time than the others. This could be due to the difficulties in carrying out the training phase using a limited number of instances of the minority class. Also for the other techniques results show an overhead, although less significant than the two mentioned above. In particular *Undersampling* performance is very close to the *No-balancing* one.

Overall, except *Undersampling*, all data balancing techniques introduce significant overhead in time consumption of ML algorithms. While the two techniques based on meta-classification (i.e., *One-Class Classifier* and *Cost-Sensitive Classifier*) take much more execution time, the other ones (i.e., *Oversampling*, *Undersampling*, and *SMOTE*) show performance pretty close to *No-balancing*.

Our findings have two clear implications for the research community. First, the problem of code smell detection is naturally unbalanced and, for this reason, *difficult to treat with machine learning techniques*: as such, researchers interested in this finding are called to devise novel effective strategies to make machine learning really suitable for code smell detection. Perhaps more importantly, the problem of code smell detection lends itself to possible *interpretation bias*: thus, we advice researchers in the field to carefully analyze the internal mechanisms of detection models before interpreting their goodness. The results have implications for both researchers and practitioners. Both are interested in understanding quantitatively the effectiveness and efficiency of applying data balancing to Machine Learning code smell detectors.

Analysing the Impact of Metric Selection

The study discussed in Chapter 4 also demonstrates that machine learning models achieve good accuracy only for some code smells, regardless of the adopted balancing technique. Hence we conducted an additional analysis to assess the effectiveness of the heuristic-based techniques. Our main goal is to investigate whether the low accuracy is due to the machine learning techniques or caused by the reduced prediction power of the used metrics.

We hypothesise that metrics with low prediction power are detrimental for both Machine Learning-based and heuristics-based approaches. This analysis was conducted only for Object-Oriented code smells where the accuracy of machine learning techniques is low.

Table 7.3: Aggregate Results for Heuristics-based and Machine-Learning-based Code Smells Detection

	Code Smells Detection Comparison							
	Precision		Recall		F-measure		MCC	
	ML	H	ML	H	ML	H	ML	H
God Class	**0.26**	0.08	0.93	**1.00**	**0.41**	0.16	**0.49**	0.28
Complex Class	**0.26**	0.23	0.65	**0.72**	**0.37**	0.35	**0.40**	0.37
Class Data Should Be Private	**0.23**	**0.23**	**0.55**	0.42	**0.33**	0.30	**0.35**	0.31
Spaghetti Code	**0.16**	0.11	0.34	**0.47**	**0.22**	0.18	**0.22**	**0.22**
Long Method	0.15	**0.57**	**0.56**	0.37	0.23	**0.44**	0.30	**0.42**
Feature Envy	0.03	**0.05**	0.44	**0.46**	0.05	**0.10**	0.11	**0.15**
Inappropriate Intimacy	**0.27**	0.04	0.15	**0.43**	**0.19**	0.07	**0.19**	0.12
Middle Man	**0.16**	0.04	**0.87**	0.43	**0.28**	0.07	**0.37**	0.12
Refused Bequest	**0.12**	0.04	0.05	**0.40**	**0.07**	**0.07**	0.07	**0.11**
Speculative Generality	0.01	**0.04**	**0.65**	0.43	0.02	**0.08**	0.02	**0.13**
Long Parameter List	**0.35**	0.04	**0.95**	0.41	**0.51**	0.08	**0.58**	0.12

Table 7.3 reports the aggregate results of the evaluation metrics for (i) the machine learning-based technique executed with the best balancing technique for each code smell (ML); (ii) the heuristic-based approach based on the detection rules described in Table 4.1 (H).

MCC values are generally low for any of the considered code smells (lower than 0.5). Except for *Long Method*, recall is always much higher than precision for heuristics-based approaches as well as for machine learning-based ones. In other words, they tend to produce a large number of false positives when these metrics are employed. Therefore, such metrics might not be adequate to discriminate smelly or non-smelly instances. Looking at these results, we note that heuristics do not outperform machine learning. On the contrary, for six of the eleven object-oriented code smells, machine learning-based approaches have a higher MCC. For instance, let us consider the case of *Long Parameter List* in which Machine Learning shows MCC equal to 0.58 that

is much higher than the one of the heuristics-based approach (i.e., 0.12). To sum up, the results indicate that the employed set of metrics (i.e., structural metrics) are not adequate in most of the cases, thus confirming previous work that deems as necessary the introduction of novel metrics as well as their combination with structural metrics to achieve better accuracy.

This motivation, led us to conduct the study presented in Chapter 5. The results reported by this study have shown that a combination of features can improve the performance of ML-based code smell detection. This was true when combining static analysis warnings raised by different automated tools, but also when combining the warnings with code metrics considered by previous work. But is this enough? To further understand this point, we have compared the performance of the proposed combined model with those of three baselines: (i) the OPTIMISTIC CONSTANT classifier, that classifies any instance as smelly; (ii) the PESSIMISTIC CONSTANT classifier, that classifies any instance as non-smelly; and (iii) a RANDOM classifier, which classifies an instance as smelly or non-smelly with a probability of 50%.

Table 7.4: Type I and Type II Errors Achieved in the comparison between the combined model, the optimistic constant, the pessimistic constant, and a random classifier

	Combined model		Optimistic Constant		Pessimistic Constant		Random	
Code Smell	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II
God Class	4034 (4.68%)	214 (0.25%)	85799 (99.53%)	0 (0.00%)	0 (0.00%)	403 (0.47%)	43156.5 (50.06%)	650.5 (0.75%)
Complex Class	4907 (7.15%)	183 (0.27%)	68375 (99.60%)	0 (0.00%)	0 (0.00%)	277 (0.40%)	34372.5 (50.07%)	26.5 (0.04%)
Spaghetti Code	5005 (5.71%)	669 (0.76%)	86886 (99.09%)	0 (0.00%)	0 (0.00%)	796 (0.91%)	44526 (50.78%)	391.5 (0.45%)
Inappropriate Intimacy	728 (1.10%)	175 (0.26%)	65879 (99.69%)	0 (0.00%)	0 (0.00%)	205 (0.31%)	33984 (51.43%)	1202.5 (1.82%)
Lazy Class	1698 (3.29%)	108 (0.21%)	51525 (99.76%)	0 (0.00%)	0 (0.00%)	123 (0.24%)	26419.5 (51.15%)	101.5 (0.20%)
Middle Man	3695 (9.10%)	62 (0.15%)	40537 (99.83%)	0 (0.00%)	0 (0.00%)	70 (0.17%)	21271.5 (52.38%)	221.5 (0.55%)
Refused Bequest	8837 (11.28%)	377 (0.48%)	77870 (99.40%)	0 (0.00%)	0 (0.00%)	467 (0.60%)	37824.5 (48.28%)	1698.5 (2.17%)

We performed this comparison in terms of Type I, that counts the number of false positive errors, and Type II, that counts the number of false negative errors. The selection of these two metrics was inspired by previous work in the literature [111]. Table 7.4 reports the total number of Type I and Type II errors. Results show that, regardless on the code smell under consideration, the PESSIMISTIC CONSTANT achieves the best results in terms of total errors, i.e., Type I + Type II, thus pointing out once again the low performance of ML-based code smell detection techniques.

These results lead to clear implications: The problem of code smell detection through machine learning still requires specific features that have not been taken into account yet. Moreover, additional AI-specific instruments should be considered in the future with the aim of improving the code smell detection capabilities of these techniques.

The subjective interpretation of code smells. One of the best-known limitations of code smell detection is the subjectivity of the results provided, as perceived by developers. In Chapter 6 we devised a novel code smell prioritization approach that is able to capture the real developer's perception of code smells in source code. The approach has been evaluated showing very good performance, thus suggesting that it could represent a good solution to overcome the limitation of the subjective interpretation of code smells. While we are confident that this is true, we still suggest researchers further investigate this limitation.

Part II

FURTHER RESEARCH ON TECHNICAL DEBT: THE TESTING PERSPECTIVE

8

BACKGROUND & RELATED WORK

8.1 INTRODUCTION AND MOTIVATION

Software testing is the activity that allows developers to check that the source code works as expected [253]. Having high-quality software is a vital requirement for developers to keep staying on the market. Over the last years, a number of researchers have investigated the properties that make test code more effective [37, 49, 102, 208, 209] as well as their relation to the ability of catching defects in production code [44, 49, 154]. Researchers have successfully demonstrated that the quality of test suites has a strong correlation with the post-release defects that appear in the production classes they test [49, 145], i.e., the higher the test quality the lower the likelihood that the corresponding production code will be affected by defects. For instance, Kochhar *et al.*[145] have shown that having a higher assertion density (measured as the number of assert statements per test lines of code) relates to a significantly lower number of defects in production code. Similarly, other studies have investigated the correlation between different types of code coverage and post-release defects metrics [37, 49] as well as test smells [280] on software quality, always reporting that test-related factors are relevant to explain the number of post-release defects in production code. It should be noted that by test-related factors we mean the set of metrics that can characterize the quality of tests, e.g., their design quality rather than their ability to cover the production code.

To provide the reader with a clearer understanding of how researchers in the past have studied the relation between the characteristics of tests and post-release defects, let consider the example reported in Listing 8.1.[1] It concerns with the test case named `testAdd`, which belongs to

1 The suite to which the test case belongs has 1,128 lines of code - we report only an exemplary test case for the sake of understandability.

Listing 8.1: Example of a test case.

```
1. @Test
2. public void testAdd() {
     3. SparseGradient x = SparseGradient.
        createVariable(0, 1.0);
     4. SparseGradient y = SparseGradient.
        createVariable(1, 2.0);
     5. SparseGradient z = SparseGradient.
        createVariable(2, 3.0);
     6. SparseGradient xyz = x.add(y.add(z));

     7. checkF0Fl(xyz, x.getValue() + y.getValue()
            + z.getValue(), 1.0, 1.0, 1.0);
}
```

the `SparseGradientTest` test suite of the Apache Commons Math 3.3 system—one of the projects considered in our study. The test aims at verifying that the `add` method of the corresponding production class `SparseGradient` correctly sums a set of numbers; to this aim, it instantiates the variables to be added (lines #3, #4, and #5 of Listing 8.1) and sums them using the `add` method (line #6). Finally, it calls the method `checkF0F1`, implemented in the same test suite, that verifies the sum and checks for the first order derivative passed as additional parameters (line #7). The test case is able to entirely cover the corresponding production method (line coverage=100%) and, similarly, the entire test suite has a line coverage of 98%. In the subsequent release of Apache Commons Math, the class `SparseGradient` did not exhibit any defect: a possible reason lies in the ability of the corresponding test suite to provide developers with an effective instrument to verify the presence of defects and, as a matter of fact, previous work in literature have discovered a correlation between code coverage of tests and post-release defects in production code, i.e., the higher the coverage the lower the number of defects in subsequent releases of system [37, 49].

Despite the effort made by the research community in understanding the relations between test quality and post-release defects, we identify a key common limitation in previous work: they analyzed the impact of various

test-related factors in isolation, controlling neither for other test-related factors nor for additional known phenomena affecting the quality (measured in terms of post-release defects) of production code (e.g., product metrics [19, 43]).

To clarify the practical effect of this common limitation, let consider again the example reported in Listing 8.1. While the code coverage was very high and suggested that the test suite could effectively help developers in spotting post-release defects, the fact that the class `SparseGradient` was actually defect-free in the subsequent release of APACHE COMMONS MATH might have and might not have been due to the high code coverage of `SparseGradientTest`. Other factors, for instance the low amount of maintenance activities performed on the production class, may have played a role. This is what actually happened to `SparseGradient`: it did not undergo any modification in APACHE COMMONS MATH 3.4 and, therefore, this was the reason making it defect-free—independently from the high value of code coverage of the corresponding test suite. Should this example be generalizable, it would mean that the findings reported in literature would not depict a clear picture on the relation between the characteristics of tests and their ability to foresee post-release defects.

As such, understanding this relation can have fundamental importance to preserve code quality and reduce the number of defects that appear in production.

Other than considering standard systems, could be even more important to assess and verify these properties in mobile applications.

Over the last years, indeed, the need for mobile applications that could connect people and support them when performing any kind of activities [298] has increased rapidly; as a matter of fact, these days we have more connected mobile devices (~7.94 billion) than people [13, 284].

The quality of mobile applications plays a central role for developers to ensure that their apps stay on the market, keep gaining users, and have a high commercial success [173, 188, 233].

Software testing is among the most relevant and well-established methods to control for source code quality [204]. Its relevance is even more critical in mobile computing [202], where continuous releases increase the risk

of introducing defects [192, 212]. Furthermore, mobile applications have peculiar characteristics, e.g., apps have multiple sensors and users interact with them through touch-screen, that make testing different and more challenging than those of traditional systems [167]. For the above mentioned reasons, the research community has been actively looking for solutions that could improve the way developers test their applications: these efforts produced the definition of several Graphical User Interface (GUI) testing approaches [101, 173, 182] and frameworks able to ease the verification of both functional and non-functional requirements [70, 93] that can be used to automate some of the developer's activities.

While these approaches have shown to be somewhat actionable, there are a number of limitations, e.g., poor ability to generate valid test data to exercise specific program executions [53], that do not allow the definition of comprehensive, effective, and practical automated testing approach [128]. These limitations make mobile developers reluctant to use automated testing tools and more prone to keep writing tests manually [128, 146, 165].

Unfortunately, the nature of manually written tests has been barely analyzed in literature: empirical studies focused on the characteristics of automated tests [62, 146, 274], while little is known on (1) the extent to which mobile applications contain manual tests, (2) how many of them can be actually executed, (3) what is their quality, considering either test code design and effectiveness metrics, and (4) what is their capabilities in foreseeing defects in production code. An improved understanding of mobile app testing from the perspective of manually written tests may provide important insights to the research community. In fact, should mobile apps be well-tested and/or manually written tests be already effective, the urgency of designing automated approaches could be toned down while focusing on how to complement manually written tests and provide developers with information useful to make tests more effective (e.g., which test data should be used to exercise certain boundary conditions). On the other side, the empirically-grounded results may serve to practitioners as an additional proof of the need for using automatic solutions as well as further supporting the testing research community.

8.2 RELATED WORK

This section analyzes and discusses the related literature about technical debt in test code from two different perspectives: (i) test-related factors affecting code quality and (ii) test code quality in mobile applications.

8.2.1 *Test-related factors affecting source code quality*

Nagappan *et al.*[209] used the Software Testing and Reliability Early Warning (STREW-J) metric suite [209] to investigate the relation between in-process testing metrics and software quality. This suite includes a variety of test metrics belonging to three categories: (1) Test quantification, e.g., presence of test cases or assertion density, (2) Complexity and OO metrics, e.g., complexity and coupling of tests, and (3) size, i.e., the lines of code of tests. Their investigation—conducted on 54 small to large industrial companies—showed a significant relation between metrics in the suite and the emergence of post-release defects. These findings were later confirmed by Rafique and Misic [261], who pointed out that these metrics are even more effective in the context of test-driven development. With respect to these papers, our aim is to contextualize their results when considering a wider set of test-related factors known in literature to impact post-release defects. At the same time, we aim to shed lights on how much the power of test-related factors increases/reduces when additional factors related to production code are taken into account.

Other studies found a relation between test effort and product quality [208, 287] based on other testing metrics such as code coverage [37, 49, 208] and other static metrics (e.g., number of assertions) [209]. Kudrjavets *et al.*[154] showed the existence of a high correlation between assertion density and defect-proneness of production code, while Catolino *et al.*[44] showed that this relation may be due to the experience of the testing teams. In the experimental setting, these papers verified the relation of the considered test-related factors to post-release defects by considering the former alone, i.e., without controlling for possible confounding factors influencing the results. As such, the setting might lead to a limited view of the phenomenon.

Chen and Wong [49] used code coverage for software failures prediction and showed that this metric influences code quality. Later, Cai and Lyu [37] confirmed this result. Nevertheless, a recent work by Kochhar *et al.*[145] contradicts those findings, reporting that coverage has an insignificant relation with the number of post-release defects. This cluster of papers shares the analysis methods employed: they relied on linear and logistic regression to understand how the considered test-related factors were correlated to the presence of defects in future software releases. Also in this case, the test-related factors were considered alone and without additional confounding factors.

Spadini *et al.*[280] and Qusef *et al.*[260] studied the relation between test smells and software quality in terms of post-release defects. The former set the problem from a statistical perspective: test smells were controlled for the presence of code smells in production code as well as additional CK metrics computed on the exercised classes. While the key results of the study showed that smelly test suites make the production code more fault-prone, Spadini *et al.*[280] did not consider the effect of test smells when other test-related factors are included. The latter first analyzed the evolution of test smells in Apache Ant; then, they used correlation analysis to study the relation between test smells and post-release defects, finding a positive correlation. This paper shares the same limitations of the other previous works, hence not considering neither other test-related nor confounding factors - which is the object of our study.

8.2.2 *Test code quality in mobile applications*

The ever increasing complexity of mobile applications, given by their peculiarities (e.g., ensuring that the application is downloadable, works seamlessly, and gives the same experience across various devices and users) as well as by their differences with respect to standard applications [319, 338], has pushed the research community to define methods to support developers with testing activities [202]. Researchers have been investigating how developers test their mobile applications in comparison to standard systems [202], showing

dissimilarities, peculiarity and possible effective practices. In this section we mainly focus on the studies aiming at analyzing testing practices of mobile developers by (i) surveying and/or interviewing practitioners [146, 165] and (ii) performing mining software repository studies [62, 146].

Linares-Vásquez *et al.*[165] surveyed 102 open-source ANDROID developers on their habits when performing testing, focusing on (i) their practices and preferences, (ii) automated testing methods employed, and (iii) perception of code coverage as indicator of test code quality. As a result, they found that developers rely on usage models (e.g., use cases, user stories) of their applications when designing test cases and perceive code coverage not necessarily important for measuring the quality of test cases. Subsequently, the same authors [167] investigated current tools and frameworks that support mobile testing practices, including benefits and trade-offs between different approaches/tools. A similar work has been done by Choudhary *et al.*[53], which benchmarked automated test input generation tools, discovering that MONKEY, the random testing tool integrated within ANDROID STUDIO is still among the best ones.

Along the same direction, the work of Kochhar *et al.*[146] surveyed 83 ANDROID developers and 27 WINDOWS app developers at MICROSOFT to study techniques, tools, and types of testing used in the mobile context. At the same time, they also analyzed 600 ANDROID apps in terms of the extent to which they are tested, assessing line and block coverage. The results showed that ANDROID apps are not properly tested (i.e., 86% do not present any test cases), and this seems to be in line with the perception of developers, who are not aware of many existing testing tools.

Erfani *et al.*[128] interviewed 191 mobile developers asking about current testing practices. Results showed that there is a lack of robust monitoring, analysis, and testing tools. The work of Silva *et al.*[274] showed similar results. Indeed, they studied 25 open-source ANDROID apps in terms of test frameworks adopted, highlighting that mobile apps are not properly tested; a possible reason behind this result may be related to the lack of effective tools [274]. A recent study by Cruz *et al.*[62] investigated working habits and challenges when testing mobile apps. In particular, they analyzed 1,000

ANDROID apps, showing that testing technologies (e.g., JUNIT) are absent in the 60% of the cases; however, when a mobile application is tested, the authors observed an increment of contributors and commit, moreover they noticed that mobile apps with tests have got an high number of minor code issues. Finally, the most recent work was performed by Lin *et al.*[164]; in particular, they conducted a large-scale analysis, over 12.000 mobile apps, to understand how test automation works in this context, i.e., tendency to write tests and practice itself. Moreover, they analyzed how test automation impacted the popularity and surveyed 148 developers to have feedback about automation test adoption.

8.3 OUR CONTRIBUTION ON TECHNICAL DEBT IN TEST CODE

This part of the thesis aims at addressing the open issues about technical debt in test code. To this aim, first of all we asked whether and to what extent, test-related factors have a real impact on software code quality, hence defining our third high-level research question (i.e., RQ_c). In this regard, in Chapter 9 we provide a multivocal literature review aiming to identify all test-related factors that have been associated to software code quality in the past. Then, to statistically verify this relation, in Chapter 10 we present a case study in which we explore how the test-related factors identified in literature are related to software quality. In this study, we build statistical models, to study how test-related factors relate to the number of post-release defects in production code, even when we also consider other product and process metrics as confounding factors. The main finding of our study is that most of the test-related factors do not have a direct relation with software quality, as opposed to factors such as production class LOCs and pre-release changes.

Other than studying technical debt from the testing perspective on standard software system, we also wondered about testing in mobile applications through the definition of the fourth high-level research question (i.e., RQ_d).

Chapter 11 reports a large-scale empirical study on the prominence, quality, and effectiveness of the tests manually written by mobile developers. The study revealed that mobile applications are not sufficiently tested, e.g., we

found just 2 test suites per app on average. Most of the available tests were at unit-level and related to the verification of the application logic, while GUI-related classes and storage of the considered apps were mostly untested. In addition, we discovered that the majority of tests have design issues, as measured by test smells, even though their metric profile would not suggest a low design quality. Finally, also test effectiveness has been proven to be low.

9

COLLECTING TEST-RELATED FACTORS: A MULTIVOCAL LITERATURE REVIEW

This chapter presents a Multivocal Literature Review (MLR) [94] aiming to identify the test-related factors that might influence the quality of the exercised production code. An MLR is an enhanced version of systematic literature reviews that not only considers *white papers*, i.e., those that have been published in conferences and journals, but also *gray documentation*, i.e., the knowledge that can be extracted from online unpublished sources like websites and blog posts. Next, we describe methodology and results achieved from the literature review.

9.1 RESEARCH METHODOLOGY

The *goal* of the multivocal literature review is to collect the test-related factors that have been analyzed and/or discussed in both previously published work and online unpublished sources, with the *purpose* of providing a comprehensive view of which factors have been associated to post-release defects. The *perspective* is that of researchers who are interested in gaining knowledge of test-related factors and their relation with software quality.

9.1.1 *Research Question*

To address the goal of our study, we set up the following research question:

RQ$_0$. *What are the test-related factors related to post-release defects, according to the available white and gray literature?*

As further reported in this section, we followed well-established research guidelines to conduct systematic and multivocal literature reviews [94, 143].

9.1.2 *Search Query Definition*

The search query represents the set of keywords that are used to search reliable sources on the phenomenon of interest [143]. In our case, we made two main considerations before defining it. First, we noticed that multiple terms could be used as synonym of *'defect'*: these are *'bug'*, *'fault'*, and *'failure'*.[1] Secondly, the term *'post-release'* could also be referred to in different ways, namely *'post-production'*, *'post-delivery'*, and *'post-verification'*. According to these considerations, we defined the following search query:

(*'test'*) **AND** (*'post-release'* **OR** *'post-production'* **OR** *'post-delivery'* **OR** *'post-verification'*) **AND** (*'defect'* **OR** *'bug'* **OR** *'failure'* **OR** *'fault'*)

9.1.3 *Selecting the Source Engines*

The selection of relevant sources is a crucial activity to provide a comprehensive description of the state of the art [94, 143]. In our context, this step consisted of selecting search engines that could cover both white and gray literature. As for the former, we selected all major databases indexing published papers: these are (1) the IEEEXplore Digital Library,[2] (2) the ACM Digital Library,[3] (3) Science Direct,[4] (4) SpringerLink,[5] and (5) Scopus.[6]

The selection of these search engines was driven by our willingness to consider as many sources as possible when conducting our literature search. These databases are widely recognized as the most representative for research in the field of software engineering [35, 133] and contain a massive amount of resources, i.e., journal articles, conference and workshop proceedings, books, etc., concerned with the research question we posed.

1 We did not include the term *'error'* since it refers to the *action* performed by a developer to introduce a defect in source code rather than to the defect itself [253].
2 Link: https://ieeexplore.ieee.org/Xplore/home.jsp
3 Link: https://dl.acm.org
4 Link: http://www.sciencedirect.com
5 Link: https://link.springer.com
6 Link: https://www.scopus.com

As for the gray literature, we followed a similar approach as other multivocal literature reviews (e.g., [95, 125]) and exploited the GOOGLE search engine.[7]

9.1.4 *Exclusion and Inclusion Criteria Definition*

Exclusion and inclusion criteria report the characteristics that a retrieved source must not (or must) have to be considered useful for addressing the research question [94, 143]. Also in this case, we needed to define criteria depending on whether a resource comes from the white or the gray literature, as some characteristics might not be applied for gray resources.

As for the white literature, we adopted the following exclusion criteria:

- Articles that were not focused on investigating the relation between test-related factors and post-release defects, e.g., papers studying how test smells relate to mutation coverage;
- Articles that have later been extended; particularly, in case of a conference paper has been extended to journal, we only considered the journal article as it is more complete.
- Articles not reporting any empirical validation of the relation between test-related factors and post-release defects, e.g., non-validated conjectures of the existence of a relation between test smells and code coverage;
- Articles that were not written in English;
- Articles whose full text was not available;
- Articles that did not undergo a peer-review process, e.g., M.Sc thesis;
- Duplicate papers retrieved by multiple databases.

We set one main inclusion criterion:

7 Link: https://www.google.com

- Articles reporting an empirical validation of the relation between test-related factors and post-release defects, e.g., papers studying how test smells relate to post-release defects.

It is worth noting that we did not set any temporal limit to our search, as we were interested in retrieving all possible sources for conducting a comprehensive analysis of test-related factors and post-release defects.

Turning the attention to the gray literature, the main challenge was represented by the assessment of the reliability of a source. Indeed, among the resources retrieved, there might be some that did not have the minimum quality to be considered reliable for our literature review. To take this aspect into account, we assessed the reliability of gray resources by relying on the guidelines provided by the University of Wisconsin[8] and, according to them, excluded unreliable resources. In particular, these guidelines are:

- *Author*. Information on the internet with a listed author is an indication of a credible site. If an author is willing to stand behind the information presented (and in some cases, include his or her contact information) is a good indication that the information is reliable.

- *Date*. The date of any research information is important, including information found on the Internet. By including a date, the website allows readers to make decisions about whether that information is recent enough for their purposes.

- *Sources*. Credible websites, like books and scholarly articles, should cite the source of the information presented.

- *Domain*. Some domains such as .com, .org, and .net can be purchased and used by any individual. However, the domain .edu is reserved for colleges and universities, while .gov denotes a government website. These two are usually credible sources for information. Websites using the domain .org usually refer to non-profit organizations which may have an agenda of persuasion rather than education.

8 Link: https://uknowit.uwgb.edu/page.php?id=30276

- *Site Design*. This can be very subjective, but a well-designed site can be an indication of more reliable information. Good design helps make information more easily accessible.

- *Writing Style*. Poor spelling and grammar are an indication that the site may not be credible. In an effort to make the information presented easy to understand, credible sites watch writing style closely.

Of course, we only considered resources written in English and that were fully available for reading. At the same time, to include a resource in our study we defined the following two criteria:

- The resource must report on practitioner's experiences and/or discussion of using test-related factors to establish the likelihood to have defects in production code;

- The resource must describe the test-related factor(s) it refers to, i.e., it must clearly mention that the test executability represents an important factor to assess post-release defects.

9.1.5 *Execution of the Multivocal Literature Review*

Once defined the ground for our multivocal literature review, we proceeded with its execution. Figure 9.1 overviews the process, reporting the inputs/outputs of each stage as well as summarizing the number of resources retrieved from each search engine and considered for our study. For the sake of understandability, the figure reports in white background the steps referring to the white literature review, while in gray the parts related to the gray literature.

The entire process was jointly executed by two of the authors in the period between May 11 to June 10, 2020. Whenever possible, the two authors met physically to perform the tasks; otherwise, they conducted the review through SKYPE. The entire execution took around 80 person/hour: the inspectors

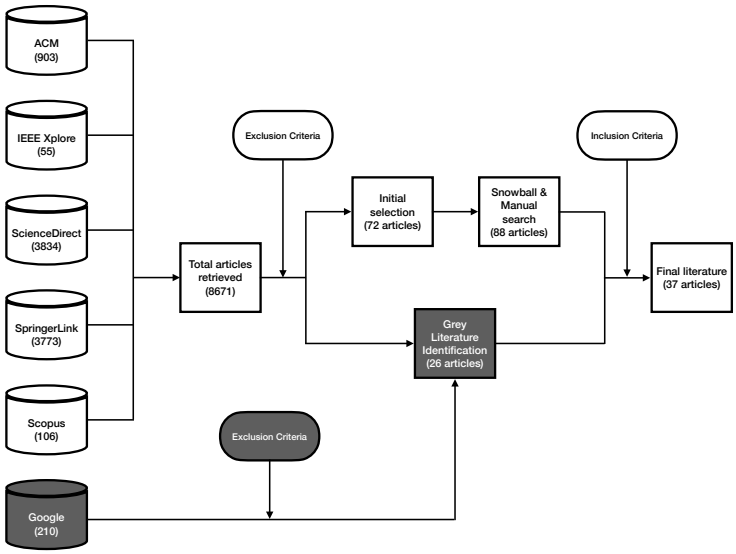

Figure 9.1: Steps performed when conducting the multivocal literature review.

started analyzing the white literature, as this was supposed to take longer. Upon completion, they then focused on the gray literature.

As shown, when executing the search query on the white literature search engines, we obtained a total of 8,671 hits: most of them came from SCIENCE DIRECT and SPRINGERLINK, which returned 3,834 and 3,773 results, respectively. The other databases were instead more restrictive when returning results.

The inspectors applied the exclusion criteria on the initial set of papers retrieved. In so doing, they mainly focused on title and abstract; nevertheless, in cases where the exclusion criteria could not be applied solely looking at these pieces of information, they explored the content of the paper more, for instance by reading the research questions, methodology, or conclusion of the paper. The exclusion criteria led to the removal of the vast majority of the initial sources (8,599 papers), giving as output a candidate set of 72 papers.

At this point, the inspectors adopted a backward and forward snowballing approach as recommended by Wohlin [326]. This was based on one iteration which consists of examining (i) the outcoming citations, namely the list of

papers cited by the article under investigation and (ii) the incoming citations, namely the list of papers citing the article under investigation, with the aim of augmenting the candidate set of papers with additional resources that were not identified through the search engines. In the first case, the inspectors went over the citations reported in the candidate papers. In the second case, they used GOOGLE SCHOLAR[9] to retrieve the list of papers citing the candidate ones. As previously done, the inspectors first focused on the title of a cited/citing paper: if it was not possible to establish the actual relevance of a new resource, they downloaded it and started reading its content. At the end of this stage, the inspectors included 16 papers to the candidate set, which reached 88 sources.

Finally, the inclusion criterion was applied: 11 papers passed the examination, forming the final set of white resources to be considered in our study.

As for the gray literature review, the inspectors followed a similar methodology. When executing the search query on GOOGLE, it returned a total of 210 results, distributed on 21 pages. It is worth noting that the inspectors performed the entire gray literature identification process using the 'incognito' mode to avoid that their personal navigation history could bias the results of the search process. Given the limited amount of results, the inspectors could browse each of them and apply the specific exclusion criteria established for the gray literature search. The joint work allowed them to immediately discuss the suitability and reliability of the resources analyzed: as an outcome, they identified a set of 26 candidate resources. Interestingly, all of them passed the inclusion criteria, composing the final set of gray resources and contributing to the grand total of 37 sources included in our multivocal literature review.

9.1.6 *Quality Assessment and Data Extraction Process*

As a final step of our multivocal literature search, we assessed the quality of the retrieved literature sources and extract data about the test-related factors associated with post-release defects.

9 Link: https://scholar.google.com

In particular, after the selection process of both white and gray literature, we defined a checklist to assess the reliability and thoroughness of the selected sources. The checklist included the following questions, which have been defined with the goal of determining whether the considered sources actually treated test-related factors and their relation with software quality:

1. Are the test-related factors mentioned in the source clearly defined?

2. When considering white papers, are the test-related factors actually assessed against post-release defects?

3. When considering white papers, is the research methodology clearly stated?

4. Are the conclusions stated supported by data (for white papers) or clear motivations (for gray sources)?

To each of these questions, the inspectors could reply with *'Yes'*, *'No'*, *'Partially'*. The inspectors considered a study as partial in cases where the methodological details could have been derived from the text, even if they were not clearly reported. These answers were scored as follows: *'Yes'*=1, *'Partially'*=0.5, and *'No'*=0. For each primary study, its quality score was computed by summing up the scores of the answers to all the four questions. At the end of the process, the inspectors classified the quality level into *High* (score = 4 for white papers, score = 3 for gray articles since they did not include point 3 of the quality assessment), *Medium* ($2 \leq$ score < 4 for white papers, score = 2 for gray literature), and *Low* (score = 1). According to our assessment, 2 resources were classified as *Medium* and 34 as *High*. Therefore, we could be able to extract data from all the selected sources.

As for the actual extraction process, the inspectors proceeded as follow. For white papers, they looked at the experimental design in order to identify (1) the test-related metrics used as independent variables and (2) the dependent variable adopted (i.e., post-release defects). Hence, in this case the data extraction was only based on these two elements. As for gray sources, the inspectors looked at the entire text to interpret the practitioner's opinions and

elicit the test-related factors they were referring to. For example, let consider the following case, which comes from the source [S03] in the list of sources reported at the end of the chapter:

"Under the assumption that tests are of good quality, [code coverage] can uncover which parts of the software have a known level of defects vs. unknown."

As reported, in the text above the practitioner refers to code coverage and how it can be used as an indicator for the location of faults in production code.

9.2 ANALYSIS OF THE RESULTS

This section summarizes the results of our multivocal literature review. It is worth remarking that we report the entire list of sources considered when performing the review at the end of this chapter.

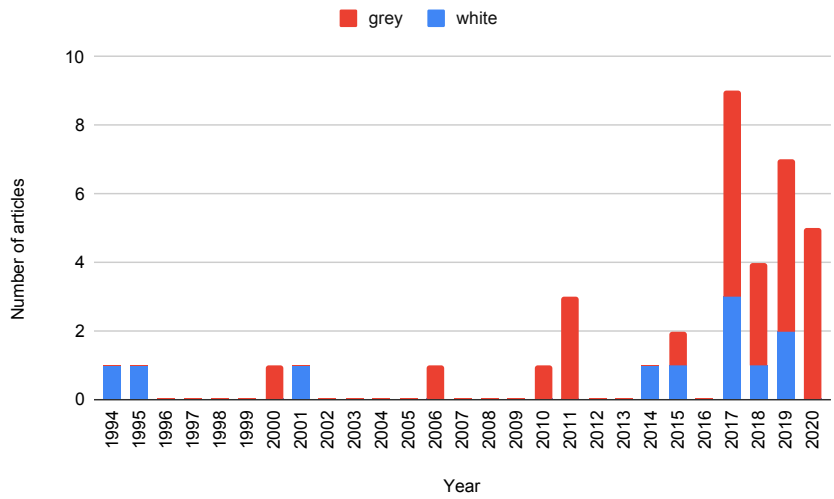

Figure 9.2: Total number of articles retrieved over years. Red bars refer to the gray resources, blue ones to white papers.

Figure 9.2 depicts the distribution over years of the resources retrieved in our review. There are two key observations to report by looking at the figure. In the first place, we observe that the number of papers published on this topic is particularly low (11) and most of them are rather recent: indeed, in the last three years we observe a slightly increasing trend. In the second place, it is surprising to see that the number of gray resources is higher than the one of white literature papers: this aspect seems to suggest that the problem of assessing the power of test-related factors to forecast post-release defects has been neglected by the research community, while it represents something important for practitioners. Also in this case, we notice that the number of gray articles increases in the last few years. Intuitively, this aspect may be due to the steady raise of development models that require the frequent execution of tests (e.g., continuous integration) and that somehow enforce developers in keeping the characteristics of tests into consideration, hence creating a growing interest into the relation between tests and software quality.

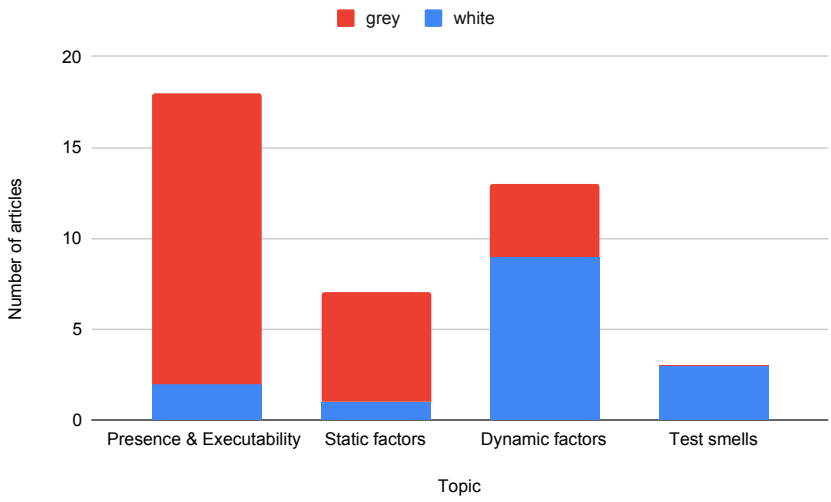

Figure 9.3: Total number of retrieved articles divided per theme. Red bars refer to the gray resources, blue ones to white papers.

Turning the attention to the types of test-related factors mentioned in the retrieved resources, Figure 9.3 overviews the themes extracted by analyzing

each of them. Interestingly, most of the gray articles mentioned *presence and executability* of test classes: in particular, a number of practitioners point out that having a properly set environment represents a crucial factor that enables the prompt identification of defects in source code. As an example, in the resource [S25], the Chief Technology Officer of the COCKROACH LABS—a well-known company that develops relational databases for cloud-native applications—reports on a two-year experience with using an open-source framework for testing distributed databases, i.e., JEPSEN.[10] He explains that an effective method to make tests actionable is to have a strong environment and, indeed, quoting from [S25]: *"every night we start up a 5-node cluster and run each test+nemesis combination for 6 minutes each"*.

Another aspect deemed as important by most practitioners is represented by *dynamic factors*, e.g., code coverage. We can notice that this type of factors has attracted most of the attention of the research community, which frequently investigated how these factors can influence post-release defects. At the same time, from the gray literature emerged the value of *static factors*, e.g., test code metrics—an aspect that seems to be overlooked by the research community. For instance, in the resource [S16], the founder of SOFTWARETESTINGMATERIAL—a blog reporting and discussing on testing practices and methodologies—reports that test code metrics can *"monitor and control process and product. [They] help to drive the project towards our planned goals without deviation"*.

Finally, the last type of test-related factor emerging from our multivocal literature review concerns with *test smells*, namely sub-optimal design or implementation solutions applied when developing test code [195]. This aspect was, however, only mentioned and investigated by researchers in the past, while we did not observe gray literature reporting on it. This possibly corroborates previous findings in the field reporting that practitioners do not perceive test smells as actual problems [302].

To conclude the discussion of the results for the multivocal literature review, Table 9.1 reports the specific metrics identified for each category of test-related factors. As shown, we identified 14 test-related factors. First, we

10 Link: https://jepsen.io

Table 9.1: Test-related factors resulting from the multivocal literature review.

Group	Name	Description
Presence and Executability	Availability of test classes	The availability of a test suite for a production class.
	Executability of test classes	The ability to run a test case for a given production class.
Static factors	TLOC	Number of lines of code of the Test Suite.
	TWMC	Weighted Method Count of the Test Suite.
	TEC	Efferent coupling of the Test Suite.
	Assertion Density	Percentage of assertion statements in the test code (i.e., number of assertions / T_LOC).
Test smells	Assertion Roulette	A test containing several assertions with no explanation.
	Eager Test	A test case testing more methods of the production target.
	Indirect Testing	A test interacting with the target via another object.
	Resource Optimism	A test that make optimistic assumptions on the existence of external resources.
	Mystery Guest	A test that use external resources (e.g., files or databases).
Dynamic factors	Line Coverage	Percentage of statement in production class that are covered by the test.
	Branch Coverage	Percentage of branches in production class that are covered by the test.
	Mutation Coverage	Percentage of mutated statement in production class that are covered by the test.

extracted metrics related to presence and executability of test classes, which are connected to the testing environment. Secondly, we identified structural metrics [50] such as test lines of code (LOC), test complexity (WMC), test coupling (EC), and assertion density. Also, we found classical metrics like line and mutation coverage [9]. Finally, we found five test smell types, i.e.,

Assertion Roulette, *Eager Test*, *Indirect Testing*, *Resource Optimism*, and *Mystery Guest*: these were the test smells associated to defect-proneness in previous work [260, 280].

Summing Up: From the multivocal literature review, we discovered four categories of test-related factors that have been associated to post-release defects in white and/or gray literature. These are connected to presence and executability of tests, static and dynamic test code metrics, and test smells.

9.3 CONCLUSION

The main contribution of this chapter is represented by the extraction of a comprehensive set of test-related factors possibly influencing post-release defects. To this aim, we conducted a multivocal literature review on the test-related factors that have been associated to post-release defects in both white and gray literature, in an effort of eliciting a comprehensive set of metrics to consider in our study.

LIST OF SOURCES

S01 - Mockus, Audris, Nachiappan Nagappan, and Trung T. Dinh-Trong. "Test coverage and post-verification defects: A multiple case study." 2009 3rd International Symposium on Empirical Software Engineering and Measurement. IEEE, 2009.

S02 - Pecorelli, Fabiano. "Test-related factors and post-release defects: an empirical study." Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2019.

S03 - Tosun, Ayse, et al. "Predicting defects using test execution logs in an industrial setting." 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). IEEE, 2017.

S04 - Qusef, Abdallah, Mahmoud O. Elish, and David Binkley. "An Exploratory Study of the Relationship Between Software Test Smells and Fault-Proneness." IEEE Access 7 (2019): 139526-139536.

S05 - Spadini, Davide, et al. "On the relation of test smells to software code quality." 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2018.

S06 - Bach, Thomas, et al. "The impact of coverage on bug density in a large industrial software project." 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, 2017.

S07 - Gren, Lucas, and Vard Antinyan. "On the relation between unit testing and code quality." 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE, 2017.

S08 - Chen, M-H., Michael R. Lyu, and W. Eric Wong. "Effect of code coverage on software reliability measurement." IEEE Transactions on reliability 50.2 (2001): 165-170.

S09 - Del Frate, Fabio, et al. "On the correlation between code coverage and software reliability." Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95. IEEE, 1995.

S10 - Malaiya, Yashwant K., et al. "The relationship between test coverage and reliability." Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering. IEEE, 1994.

S11 - Kochhar, Pavneet Singh, Ferdian Thung, and David Lo. "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems." 2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER). IEEE, 2015.

S12 - Neha, "How To Perform Post-Release Testing Effectively And Minimize Impact Of The Release To Live Clients." https://www.softwaretestinghelp.com/post-release-testing, 2020.

S13 - “Post Release Testing.” https://www.professionalqa.com/post-release-testing, 2020.

S14 - Harekal, Divakar, and V. Suma. "Implication of Post Production Defects in Software Industries." International Journal of Computer Applications 975 (2015): 8887.

S15 - Pavneet Singh Kochhar, David Lo, Julia Lawall, Nachiappan Nagappan. Code Coverage and Postrelease Defects: A Large-Scale Study on Open Source Projects. IEEE Transactions on Reliability, Institute of Electrical and Electronics Engineers, 2017, 66 (4), pp.1213 - 1228. 10.1109/TR.2017.2727062. hal-01653728.

S16 - Rajkumar, "Software Test Metrics – Product Metrics & Process Metrics" https://www.softwaretestingmaterial.com/test-metrics/, 2018.

S17 - "What Are Test Metrics?" https://www.sealights.io/agile-testing/test-metrics/, 2020.

S18 - Antinyan, Vard, et al. "Mythical unit test coverage." IEEE Software 35.3 (2018): 73-79.

S19 - King, "Tester’s Diary: Getting Ahead With Post-Release Testing" https://blog.gurock.com/post-release-testing/, 2019.

S20 - Hallowell, "Six Sigma Software Metrics, Part 1" https://www.isixsigma.com/tools-templates/software/six-sigma-software-metrics-part-1/, 2020.

S21 - Peters, "Product Managers, do you know how much your bugs cost?" https://deanondelivery.com/product-managers-do-you-know-how-much-your-bugs-cost\-72b6e36e7684, 2018.

S22 - Elish, Mahmoud O., and David Rine. "Design structural stability metrics and post-release defect density: An empirical study." 2006 30th

Annual International Computer Software and Applications Conference (COMPSAC'06 Supplement). IEEE, 2006.

S23 - Vinke, L., P. Klint, and M. Pil. "Estimate the post-release Defect Density based on the Test Level Quality." (2011).

S24 - Rothman, Johanna. "What does it cost you to fix a defect? and why should you care." Retrieved March 13 (2000): 2010.

S25 - Darnell, "Lessons Learned from 2+ Years of Nightly Jepsen Tests" https://www.cockroachlabs.com/blog/jepsen-tests-lessons/, 2019.

S26 - Bach, Thomas, et al. "The impact of coverage on bug density in a large industrial software project." 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, 2017.

S27 - "How to measure Defect Detection Efficiency/Rate?" https://club.ministryoftesting.com/t/how-to-measure-defect-detection-efficiency-rate/15313, 2018.

S28 - Sridharan, "Testing in Production, the safe way" https://medium.com/@copyconstruct/testing-in-production-the-safe-way-18ca102d0ef1, 2017.

S29 - Sharma, "Why Are Bug Tracking Tools so Important for Testing Teams?" https://dzone.com/articles/why-is-bug-tracking-tool-so-important-for-the-test, 2019.

S30 - "Root Cause Analysis" http://www.helpingtesters.com/root-cause-analysis/, 2017.

S31 - Capgemini, "Capgemini's Quality Blueprint" https://www.capgemini.com/br-pt/wp-content/uploads/sites/8/2017/07/Capgemini___s_Quality_Blueprint.pdf, 2011.

S32 - "Defect Metrics – An indicator of quality of product under test" https://qainfotech.com/defect-metrics-an-indicator-of-quality-of-product\-under-test/, 2010.

S33 - "Quality metrics: Defect tracking throughout the software lifecycle" https://searchsoftwarequality.techtarget.com/tip/Quality-metrics-Defect-tracking-throughout-the-\software-lifecycle, 2011.

S34 - Cummings-John, "How continuous testing supercharges your development process" https://techbeacon.com/app-dev-testing/use-continuous-testing-supercharge-your-development\-process, 2019.

S35 - Riley, "Don't Hate the Tester, Hate the Test Process" https://applitools.com/blog/dont-hate-the-tester-hate-the-test-process/, 2020.

S36 - Starling, "You don't have enough tests and you never will!" https://www.bugsnag.com/blog/better-software-testing, 2017.

S37 - "What to do when defect is found in production but not during the QA phase?" https://sqa.stackexchange.com/questions/27680/what-to-do-when-defect-is-found-in-production-but-\not-during-the-qa-phase, 2017.

10

STUDYING THE RELATION BETWEEN TEST-RELATED FACTORS AND POST-RELEASE DEFECTS

In this chapter, we present a case study conducted on eight systems of the Apache ecosystem in which we explore how test-related factors are related to software quality. As done in previous studies, we operationalize software quality at software component level (in our case, file level) and by measuring the component's post-release defects. To this aim, we build statistical models to study how test-related factors relate to, i.e., (i) presence and executability of test suites, (ii) statically computable test code factors, and (iii) dynamically computable test code factors, relate to the number of post-release defects in production code, when also considering several product and process metrics.

10.1 RESEARCH METHODOLOGY

The *goal* of the empirical study is to investigate how test-related factors are related to post-release defects, with the *purpose* of measuring the explanatory power of having a good test suite on software code quality. The *perspective* is of both researchers and practitioners: the former are interested in assessing the relation between test-related factors and post-release defects, while the latter are interested in understanding the extent to which good quality test suites can help them to reduce post-release defects.

10.1.1 *Research Questions and Methodological Sketch*

Our study is structured around three main research questions. Figure 10.1 synthesizes the levels of our analyses. In an ideal scenario, all production classes in a software project have corresponding tests that can be successfully

executed. However, in practice this is rarely the case, which represents an interesting scenario for us, because it creates a *natural experiment* where we can compare software quality (measured as post-release defects) in both tested and untested classes. Therefore, we started our empirical study by investigating the role played by the presence of test classes and their executability with respect to software quality.

RQ$_1$. *How do presence and/or executability of test classes relate to post-release defects of production code?*

We further investigated the role of testing on source code quality at a finer level of granularity. We restricted our analysis to those test-related factors that can be statically computed (e.g., test size)

RQ$_2$. *How do statically computable test code factors relate to post-release defects of production code?*

The final part of our empirical study was focused on the understanding of how dynamic factors (e.g., code coverage [9]) relate to software quality. This maps a scenario in which production classes are exercised by executable tests.

RQ$_3$. *How do dynamically computable test code factors relate to post-release defects of production code?*

From an experimental design perspective, for each research question we established a set of *subjects*, *factors*, and *treatments* - as also illustrated in Figure 10.1. As for **RQ**$_1$, we considered all the classes of the considered systems as *subjects*, taking into account the presence and executability of tests as *factors*, with the aim of assessing the extent to which they affect source code quality (i.e., the *treatment*). In **RQ**$_2$, we narrowed the *subjects* of the study so that we could only consider the classes having at least one test associated to compute *factors* computable using static analysis (i.e., metrics and code quality indicators) and assess their impact on post-release defects. Finally, in **RQ**$_3$, the *subjects* are represented by the set of classes having at

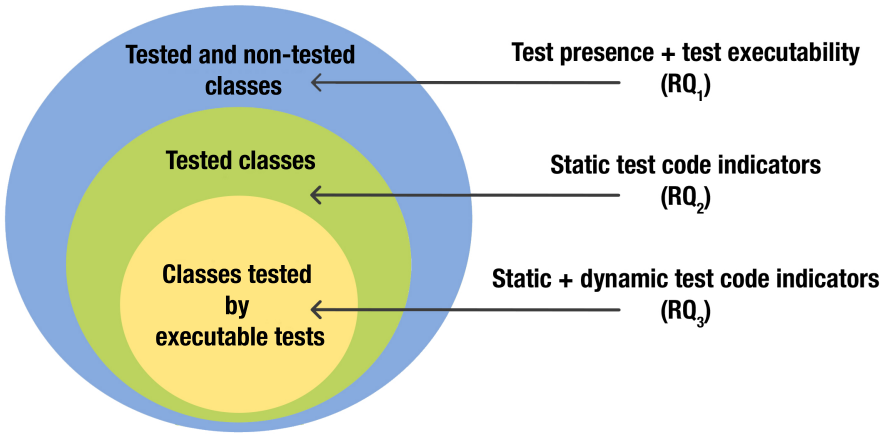

Figure 10.1: The three levels of investigation considered in our empirical study, by research question.

least one executable test, while the *factors* are the dynamically computable metrics: these are used to establish a relation between dynamic test factors and software quality. The following sections report on the subject dataset as well as the steps conducted to answer our **RQ**s.

Table 10.1: Systems from Apache Commons considered in the study

Name	# Commits	# Releases	# Contributors	# Production Classes	# Test Classes	# Defects
Codec	1,792	45	39	26	31	134
Collections	3,091	49	51	270	139	341
DBCP	1,983	62	34	53	21	367
DbUtils	656	29	21	25	20	56
IO	5,400	54	58	100	47	281
Lang	2,141	89	140	119	98	634
Math	6,395	65	34	804	404	684
Pool	1,879	168	38	41	14	205

10.1.2 *Context selection*

The *context* of the study consisted of eight open-source software systems, whose characteristics are shown in Table 10.1. Their selection was driven by

three main requirements of both our tooling and data analysis procedures. In the first place, we focused on Java because most of the test-related factors experimented can be only computed for this programming language (e.g., test smells are only defined and validated by our community for Java [308]). Secondly, we restricted our analyses to systems having a large change history information, as it is our willingness to have projects with (i) a number of post-release defects to allow significant analyses—we excluded systems with less than 50 defects reported in the corresponding issue tracker[1]—and (ii) a number of previous changes—we excluded systems with less than 500 commits—that can be used to control for our results, as detailed in Section 10.1.5. Finally, we required the selected systems to have at least one compilable release to use for our study [303], as some of the considered test-related factors can only work with compilable source code (e.g., dynamic information like line and mutation coverage). Moreover, to allow the application of the state-of-the-art tool for line and mutation coverage (i.e., PITEST), we checked for projects having no sub-modules and built with Maven, using the default Maven directory structure (i.e., the production classes are located in the 'src/main' package, while the test classes are in the 'src/test' package). As a result of this selection procedure, we found a family of eight projects that met all the aforementioned requirements, i.e., APACHE-COMMONS.

In an effort of providing more details about the context of our study, as recommended by a recent work [33], we manually dived into the information reported in the repositories analyzed, particularly looking at (1) contribution guidelines, which may indicate some relevant information of how contributors are supposed to perform testing as well as on the development process in place and (2) the change history, which reports data related to frequency of releases, number of contributors, and so on. In cases where contextual information were not minable looking at the above mentioned sources (e.g., testing type), we also analyzed online documentation (e.g., guidelines of the APACHE SOFTWARE FOUNDATION) or directly looked at the source code and extracted the remaining data. As a result, we discovered that the eight selected projects share similar characteristics as well as a similar development community—as

1 We retrieve the link to the issue tracker adopted by a certain project through the analysis of the developer's contribution guidelines.

somehow expected, since they belong to the same family of projects. APACHE COMMONS is a *commit-then-review* community, so developers who want to contribute should follow APACHE's code of conduct[2] and announce their intentions and plans on the developers mailing list before committing code. Releasing a new version of the system requires the vote of a PMC (Project Management Committee) according to the APACHE Release Creation Process.[3] For this reason, new code is not released at regular time intervals but its release depends on several aspects. Tests are written and committed together with production code indicating that developers adopt a *test-as-you-write* development strategy. All the tests available in the considered systems are written at unit-level, following a black box strategy.

10.1.3 *Dependent Variable*

The dependent variable of our study is the number of post-release defects. To compute it, we first determined whether a commit fixed a defect. This was done by employing the textual-based approach proposed by Fischer *et al.*[80], which is based on the analysis of commit messages. Such an approach has been extensively used in the past [130, 141] and was assessed to have an accuracy close to 80% [80, 236]. Specifically, we searched for issue IDs in commit messages by finding matches with the prefix used in the bug tracker system. Once retrieved a commit referencing an issue, we queried the apache issue tracker system's APIs in order to filter only issues related to resolved bugs. Then, we searched for keywords indicating fixing activities in the commit message, such as *'bug'*, *'fix'*, or *'defect'*, in order to select only the bug-fixing commits. Once we detect all bug fixing commits, we employed the SZZ algorithm [278] to obtain the commits where the defect was introduced. In particular, the SZZ algorithm relies on the annotation/blame feature of versioning systems [278]: given a defect-fix activity identified by the defect ID k, the approach works as follows:

2 http://www.apache.org/foundation/policies/conduct.html
3 https://infra.apache.org/release-publishing.html

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

- Starting from the revision *rel-fix*$_{i,k} - 1$, for each source line in f_i changed to fix the defect k, we identified the production class C_j to which the changed line belongs. Furthermore, the `blame` feature of `Git` is used to identify the revision where the last change to that line occurred. In doing that, blank lines and lines that only contain comments are identified and excluded using an island grammar parser [199]. This produces, for each production class C_j, a set of $n_{i,k}$ defect-inducing revisions *rel-defect*$_{i,j,k}$, $j = 1 \dots n_{i,k}$. Thus, more than one commit can be indicated by the SZZ algorithm as responsible for inducing a bug.

Once retrieved the list of defect-inducing commits, we computed post-release defects of a class as the number of defect-inducing activities involving the class in the period after the selected release r_j. To compute the dependent variable, we use the SPADINI2018PYDRILLER framework [279], which implements the SZZ algorithm. Some recent work has reported that the SZZ algorithm has a low accuracy, in particular concerning precision [59, 265]; should these observations be verified on our dataset, this would threat the validity of our results. For this reason, we manually validated the performance of the SZZ algorithm on our dataset. Specifically, two of the authors (the *inspectors*) jointly analyzed all the 251 total defect-inducing commits output by SZZ. In doing so, they relied on the source code of both defect-fixing and defect-inducing versions of the projects: the task was performed to understand whether a change in the defect-inducing version actually introduced the defect that was fixed in the defect-fixing version. The inspection pointed out a precision of 92% (231 correct defect-inducing commits): this result diverges from previous findings [265] and suggests that the performance of the algorithm is strongly dependent on the considered projects. As a

consequence of this validation, we excluded the 20 false positives from the analysis to have a more accurate dataset.

10.1.4 *Independent Variables*

We defined a different set of independent variables for the three **RQ**s.

Independent variables for RQ_1. In the first research question, we defined two independent variables. The first one was represented by the presence of a test class for each production class of the selected systems. We defined a variable named *'is-tested'* that assumes the value *'true'* if the production class has a test class that exercise it, *'false'* otherwise. With respect to the selection of this independent variable, there are two observations to be done. Intuitively, the presence of test classes alone cannot affect software quality—having a test does not imply the identification of defects. However, having them is a *necessary* condition: the conjecture behind the selection of *'is-tested'* as independent variable was that the availability of a test suite may allow developers to promptly identify defects, hence affecting the number of post-production defects. Hence, we conjectured an indirect relation and aimed at verifying it. In this respect, it is also worth noting that the presence of tests was one of the factors mentioned by practitioners when analyzing the gray literature: this means that the availability is actually perceived as a relevant factor for software quality. This supports the idea of an indirect relation of this factor with post-release defects.

A key point for the computation of the independent variable was related to linking each test class to each of the considered production class. All the selected projects rely on MAVEN as build tool. Thus, to perform the linking, we relied on the `pom` file, which contains the rules to identify the test classes to execute when the projects need to be built or packaged. In particular, we first identified all production and test classes by scanning the `pom` file and looking for the `sourceDirectory` and `testSourceDirectory` fields, that indicate the location of production and test code, respectively. When the fields were not reported explicitly, we considered the default source and test directories. Afterwards, we used a pattern matching approach based on naming

conventions to find the production class related to a certain test class, as it has been done in previous work [104, 171, 302]: given the name of a production class (e.g., '`ClassName`') belonging to the `sourceDirectory` folder, it checks for the presence of a test class having the same name as the production class but with the prefix or postfix "Test" in the `testSourceDirectory` (e.g., '`ClassNameTest`' or '`TestClassName`'). In case the approach cannot identify a test for a certain class, the variable *'is-tested'* for the considered production class is "false", "true" otherwise. The accuracy of this linking approach has been previously assessed [310]: it showed an accuracy close to 85% and is comparable with more sophisticated (but less scalable) techniques (e.g., slicing-based approaches [259]).

The second independent variable is named *'are-tests-executable'*: it assumes the value *"true"* if the tests exercising the production class can be ran, *"false"* otherwise. Also in this case, the selection was supported by the results achieved when analyzing the gray literature: indeed, multiple times practitioners reported that having tests that can be actually run against the production code is an important factor to assess post-release defects. In this sense, we aim at providing evidence of the effect of having runnable tests on software quality. To determine the value of the variable, for each considered project we ran the `mvn verify` command, which executes all the available tests. If the execution of a test proceeds without errors,[4] then we considered the test as executable. Otherwise we marked it as non-executable.

With respect to the executability of tests, it is worth noting that we considered the building environment directly used by developers of the considered APACHE projects. In other words, the methodology we used to run tests is exactly the same as the one used by the actual developers. Hence, we considered a real-case scenario of use of the tests, i.e., we considered a *natural experiment* that considers what actually happens in practice.

Independent variables for RQ_2. In the second research question, we studied the relation between static test-related factor and post-release defects. So, we considered test-related factors that can be computed without executing

4 Note that a MAVEN error explicitly indicates that the test cannot be run for some reasons, as opposed to a failure, which instead reports that the test has found an anomalous behavior in the production code.

test classes: these are test code metrics and smells. To compute the former, we relied on the tool by Spinellis [282]. As for the latter, we used the code-metrics based tool developed by Bavota *et al.* [25]. The detector is able to identify instances of five test smell types, namely *Mystery Guest*, *Resource Optimism*, *Eager Test*, *Assertion Roulette*, and *Indirect Testing*. All the considered smells have been related to defect-proneness by previous work in the field [260, 280]. The selection of the test smell detector was driven by the high accuracy it showed in previous studies, with F-Measure close to 86% [25, 239, 241].

Independent variables for RQ$_3$. In the context of the third research question, we also included the metrics that can be computed only when executing the test classes: the dynamic factors (i.e., Line Coverage, Branch Coverage, and Mutation Coverage). As for line and branch coverage, we used COBERTURA.[5] For mutation coverage, we used PITEST.[6] It is worth noting that the choice of using PITEST was not only driven by the fact that this is the state-of-the-art tool for mutation testing [104, 161], but also by the relatively low number of equivalent mutants, i.e., mutants having a semantically similar behavior [2], it generates: indeed, a recent study by Fernandes *et al.*[79] reported that only 20% of the mutants generated by PITEST can be considered equivalent, which is substantially lower than other mutation testing tools available in literature.

10.1.5 *Confounding Factors*

In addition to the test-related factors we collected from our MLR (see Chapter 9), the number of post-release defects may be due to other factors related to the structure of production code [19]. Thus, to avoid a biased interpretation of the results, we introduced a set of well-known source code and process metrics as confounding factors, which are summarized in Table 10.2. All of them have been previously related to defect-proneness, as further explained in the following:

5 Link:https://cobertura.github.io/cobertura/
6 Link: http://pitest.org

- We considered the **Lines Of Code (PLOC)** metric, that measures the size of production classes. According to previous findings [149, 242, 337], the larger a class the higher its fault-proneness. As such, the number of post-release defects might be a reflection of the production code size and, therefore, we computed LOC to control our findings on the impact of the presence of test suites. To measure PLOC, we used the tool devised by Spinellis [282].

Table 10.2: List of confounding factors used in the study.

Group	Name	Description
Static factors	PLOC	Number of lines of code of the Production Class
	PWMC	Weighted Method Count of the Production Class
	PEC	Efferent coupling of the Production class
Code smells	God Class	A class having a large size, poor cohesion, and several dependencies with other data classes of the system
	Class Data Should Be Private	A class exposing its attributes, thus violating the information hiding principle
	Complex Class	A class presenting a overly high cyclomatic complexity
	Functional Decomposition	A class implemented as a function
	Spaghetti Code	A class that exhibit a functional-style programming structure, declaring a number of long methods without parameters
Process Metrics	Pre-release Changes	Number of changes involving the Production class before the release date of the considered snapshot

- We computed **Weighted Method per Class (PWMC)** [50] as a way to measure the complexity of production code. A number of previous

studies has shown the metric to be related to the number of defects in which a production class will incur [68, 210, 341]. The tool by Spinellis [282] was used to compute the metric on our dataset.

- We measured the **Efferent Coupling (PEC)** of production classes because, as reported by previous research [19, 65, 91, 144, 272], the higher the coupling of a class the higher its fault-proneness. Also in this case, we employed the tool by Spinellis [282] to compute PEC.

- We considered **code smells**, i.e., symptoms of the presence of poor implementation choices [34, 88], since they are reported to be connected to the fault-proneness of production code [64, 116, 137, 225, 226, 243, 304]. We considered five code smells from the catalog by Fowler [88] that have different characteristics, namely *God Class*, *Class Data Should Be Private*, *Complex Class*, *Functional Decomposition*, and *Spaghetti Code*. These code smells have been analyzed by previous work studying their effect on source code defect-proneness [137, 226]. Therefore, our selection was driven by these findings. As for the actual detection of these code smells, we relied on DECOR [197], a state-of-the-art detection tool which has shown an accuracy close to 80% [197]. In our work we re-evaluated the precision of DECOR. The two authors previously involved in the validation of the test smells also conducted this analysis: they manually validated all the 137 code smell instances output by the tool. The task was to understand whether a certain code smell candidate given by DECOR actually revealed the existence of a design problem in source code. After the first assessment, the two inspectors compared their evaluations, reaching an agreement of 95%. The remaining 5% of cases (i.e., seven code smell candidates) were discussed and, finally, four of them turned to be real code smells. Following this validation, we (i) confirmed the good accuracy and the suitability of DECOR in our context and (ii) excluded the false positive smells from our analysis.

- We computed the number of **pre-release changes** and **pre-release defects** because metrics capturing the previous history of production

classes can reveal relevant evolution aspects [119, 262]. To compute the number of pre-release changes, we mined the change log of the considered projects and count how many times a certain production class has been modified. As for the pre-release defects, we relied again on the SZZ algorithm implemented in PyDriller [279].

10.1.6 *Statistical Modeling and Data Analysis*

After collecting the data for all the considered projects, we defined three groups of hypothesis, related to the three research questions.
As for $\mathbf{RQ}_1$, we defined two pairs of null (Hn) and alternative (An) hypotheses:

Hn1. There is no correlation between the presence of test classes and software quality, as measured by post-release defects.

An1. There is a correlation between the presence of test classes and software quality, as measured by post-release defects.

Hn2. There is no correlation between the executability of test classes and software quality, as measured by post-release defects.

An2. There is a correlation between the executability of test classes and software quality, as measured by post-release defects.

Regarding $\mathbf{RQ}_2$, we defined the following hypotheses:

Hn3. There is no correlation between test code metrics and software quality, as measured by post-release defects.

An3. There is a correlation between test code metrics and software quality, as measured by post-release defects.

Hn4. There is no correlation between test smells and software quality, as measured by post-release defects.

An4. There is a correlation between test smells and software quality, as measured by post-release defects.

Finally, we report below the hypotheses to address $\mathbf{RQ}_3$:

Hn5. There is no correlation between code coverage metrics and software quality, as measured by post-release defects.

An5. There is a correlation between code coverage metrics and software quality, as measured by post-release defects.

Hn6. There is no correlation between mutation coverage and software quality, as measured by post-release defects.

An6. There is a correlation between mutation coverage and software quality, as measured by post-release defects.

To test the hypotheses, we built a statistical model relating the independent and confounding factors to the post-release defects. A first key design decision regarded the proper choice of the statistical approach to fit our observations. We built a Generalized Linear Model (GLM) [213]. This method models the relationship between a scalar response (i.e., number of post-release defects in our case) and one or more explanatory variables (i.e., the selected set of independent and confounding factors) by fitting a linear function whose unknown model parameters are estimated from the data. We use the *'Gaussian'* family when implementing the model.

We relied on this approach for two main reasons. First, it simultaneously analyzes the effects of both confounding and independent variables on the response variable [112]. Second, it does not require distribution of data to be normal: indeed, in our case, the Shapiro-Wilk test [270] rejects the null-hypothesis, i.e., our data is not normally distributed.

To avoid multicollinearity [217], which may bias the interpretation of the results [201, 217], we first applied a hierarchical clustering based on the Spearman's rank correlation coefficient [281] of the studied variables (done using the `varclus` function available in the `R` statistical toolkit[7]), then, if two variables had a correlation higher than 0.6, we excluded the more complex one from the model. We interpreted the output of the Generalized Linear Model

7 https://bit.ly/2YFltBU

by considering the statistical significant codes it assigns to each explanatory variable, i.e., if a certain variable is deemed as statistically significant, the chances of the effect on the number of post-release defects being random is sufficiently low. We also computed the Adjusted R-squared [73] to assess the goodness of fit of the model, a metric indicating how close the data is to the fitted regression line.

10.2 ANALYSIS OF THE RESULTS

For each **RQ**, we build the models in a progressive manner, so that we can measure the explanatory power of different factors step-by-step. In particular, for each analysis we define three models: (i) the first only considers the effects of test-related factors, (ii) the second considering both production and test metrics, and (iii) a full model that includes production metrics, test-related factors and the selected process metric.

10.2.1 $\boldsymbol{RQ}_1$. *The presence and executability of tests*

Table 10.3 reports the results of our first analysis. From the total set of variables employed within the model, we had to exclude (i) PWMC and PEC because they were too correlated with PLOC and (ii) the one signaling the presence of the *Spaghetti Code* smell, which in this case had high correlation with the presence of *Blob* instances. In conclusion, the model was composed of a total of seven metrics whose distributions are described in Table 10.4. The Adjusted R-squared measured 0.264: the value can be considered "moderate" [51, 57], namely the statistical model can fit the data with a moderate precision: $\approx 85\%$ of variation is still unexplained. Results of the first model led us to reject the two null hypotheses related to our first research question (i.e., Hn1 and Hn2) in favor of the alternative hypotheses (i.e., An1 and An2), indicating that the presence and the executability of test classes have a correlation with the number of post-release defects. However, when adding confounding factors, the hypotheses cannot be rejected nor confirmed.

Table 10.3: Results for $\mathbf{RQ}_1$ - The impact of the presence and executability of tests on the number of post-release defects. $N = 1,457$

	Test			Test + Prod.			Full		
	Estimate	**S.E.**	**Sig.**	**Estimate**	**S.E.**	**Sig.**	**Estimate**	**S.E.**	**Sig.**
Intercept	0.11	0.05	*	-0.03	0.05		-0.17	0.04	***
is-tested	0.57	0.12	***	0.14	0.12		-0.09	0.11	
are-tests-executable	-0.49	0.13	***	-0.23	0.12	.	-0.05	0.11	
PLOC				0.00	0.00	***	0.00	0.00	
isGodClass				0.24	0.16		0.17	0.15	
isClassDataShouldBePrivate				0.39	0.38		0.70	0.34	
isComplexClass				-1.12	0.30	***	-0.29	0.27	
pre-release changes							0.05	0.00	***
pre-release defects							0.10	0.01	***
Multiple R-squared: 0.268; Adjusted R-squared: 0.264									
significance codes: '***'p <0.001, '**'p <0.01, '*'p <0.05, '.'p <0.1									

Table 10.4: Descriptive statistics for variables used in $\mathbf{RQ}_1$. $N = 1,457$

Variable name	**Minimum**	**Maximum**	**Mean**	**SD**
PLOC	2.00	6291.00	211.00	359.90
isGodClass	0.00	1.00	0.10	0.30
isClassDataShouldBePrivate	0.00	1.00	0.01	0.09
isComplexClass	0.00	1.00	0.02	0.14
is-tested	0.00	1.00	0.49	0.50
are-tests-executable	0.00	1.00	0.38	0.49
pre-release changes	0.00	201.00	7.32	12.26
pre-release defects	0.00	35.00	1.03	3.37
post-release defects	0.00	23.00	0.20	1.33

Looking at the table, the variable which mostly influences post-release defects is the number of *pre-release changes*. Thus, we can confirm previous findings in the field [106, 142] on the relevance of this variable: the information coming from the past history of a class is a valuable predictor of its future quality. At the same time, the contribution given by this metric somehow hides the value of other product-based confounding variables: more specifically, factors like production code size and code smells—that were found to be highly relevant to explain the future defect-proneness of source code [137, 206,

226]—are subsumed by this change history-based metric. Indeed, observing the results of the other two models which do not contain the process metric, we notice that some aspects related to the production code result to be highly significant (i.e., PLOC, Complex Class).

In the full model, the two independent variables selected for $\mathbf{RQ}_1$, i.e., *is-tested* and *are-tests-executable* are not statistically related to the number of post-release defects that will incur in source code classes. This result suggests that the mere existence of tests and/or their executability does not affect the number of post-release defects in the exercised production code.

From another perspective, our results can be also interpreted as a sign that the *quantity* of tests is not enough, and that perhaps their *quality* can serve as better indicators of post-release defects. In the next research question, we investigate whether the *quality* of tests is related to post-release defects.

Summing Up: Neither the executability nor the presence of tests are statistically significant variables to explain post-release defects when other confounding factors are taken under consideration. We confirm that the number of pre-release changes has the highest explanatory power.

10.2.2 *$\mathbf{RQ}_2$. The impact of static test code indicators*

While in the first research question we considered the entire set of instances belonging to our dataset, in $\mathbf{RQ}_2$ we have to consider on a smaller set of cases (as also seen in Figure 10.1), because we focus on the relation of statically computable test-related indicators to post-release defects, therefore the presence of tests is required to compute these indicators. The dataset we consider for the second research question has 774 observations and 184 post-release defects (as opposed to the 231 of the dataset used in $\mathbf{RQ}_1$). The descriptive statistics for all the variables are reported in Table 10.5. When removing untested classes from the dataset, the mean of the considered process metric (i.e., *'pre-release changes'*) increases (see Tables 10.4 and 10.5). This may suggest that the number of changes tends to be lower when tests are not available, perhaps because developers are less confident with

Table 10.5: Descriptive statistics for variables used in $\mathbf{RQ}_2$. $N = 774$

Variable name	Minimum	Maximum	Mean	SD
PLOC	13.00	6291.00	311.00	460.00
isGodClass	0.00	1.00	0.17	0.38
isClassDataShouldBePrivate	0.00	1.00	0.01	0.12
isComplexClass	0.00	1.00	0.04	0.19
TLOC	5.00	2210.00	168.00	248.10
Assertion Density	0.00	0.83	0.19	0.15
isAssertionRoulette	0.00	1.00	0.87	0.34
isEagerTest	0.00	1.00	0.61	0.49
isMysteryGuest	0.00	1.00	0.07	0.26
isResourceOptimism	0.00	1.00	0.02	0.14
isIndirectTesting	0.00	1.00	0.06	0.24
pre-release changes	1.00	201.00	10.00	16.21
pre-release defects	0.00	35.00	1.73	4.54
post-release defects	0.00	23.00	0.29	1.65

modifying the source code in such a circumstance or because tests are added when more changes are needed on certain files.

The multicollinearity analysis led us to remove (i) PWMC, PEC, TWMC, and TEC, as they were too correlated with the PLOC and TLOC metrics, and (ii) the variable related to the *Spaghetti Code* smell, as this still has a high correlation with the *Blob* code smell. Therefore, the model was composed of a total of 12 variables and reached an Adjusted-R squared of 0.282 (moderate).[8] This means that the model only explains $\approx 15\%$ of variation.

In the first place, the results, reported in Table 10.6, show that TLOC, is highly significant when test-related factors are considered in isolation, thus allowing to reject Hn3 in favor of An3. However, this relation cannot be extended to the full model since the addition of confounding factors led to

8 Note that the three statistical models for the different research questions are not comparable in terms of R^2, since they operate on different datasets (see Figure 10.1). We report the values for the R^2 only to give an idea of the statistical models' explanatory power.

Table 10.6: Results for $\mathbf{RQ}_2$ - The impact of static test-related factors on the number of post-release defects. $N = 774$

	Test			Test + Prod.			Full		
	Estimate	**S.E.**	**Sig.**	**Estimate**	**S.E.**	**Sig.**	**Estimate**	**S.E.**	**Sig.**
Intercept	-0.02	0.18		-0.08	0.18		-0.19	0.16	
TLOC	0.00	0.00	***	0.00	0.00		-0.00	0.00	
Assertion Density	0.16	0.44		0.02	0.43		-0.38	0.39	
isAssertionRoulette	-0.06	0.20		-0.04	0.19		0.05	0.17	
isEagerTest	0.12	0.14		0.03	0.14		-0.05	0.13	
isMysteryGuest	-0.19	0.28		-0.20	0.28		-0.54	0.25	*
isResourceOptimism	0.73	0.53		0.54	0.52		-0.22	0.47	
isIndirectTesting	-0.01	0.27		-0.04	0.27		0.09	0.24	
PLOC				0.00	0.00	***	0.00	0.00	
isGodClass				0.36	0.23		0.32	0.21	
isClassDataShouldBePrivate				0.27	0.56		0.93	0.50	.
isComplexClass				-0.97	0.41	*	-0.22	0.38	
pre-release changes							0.03	0.00	***
pre-release defects							0.09	0.02	***
Multiple R-squared: 0.294; Adjusted R-squared: 0.282									

significance codes: '***'p <0.001, '**'p <0.01, '*'p <0.05, '.'p <0.1

a decrease of the significance of test code metrics, namely Hn3 cannot be rejected nor confirmed.

The results of the full model confirm the previous ones: there is a strong relation between the process metric we considered (i.e., pre-release changes) and post-release defects even when adding factors quantifying the properties of test code. On the one hand, this finding reinforces the idea that the change process underwent by production classes is among the most valid indicators for software quality. On the other hand, statically computable properties of test code do not impact the future defect-proneness of production classes.

The only exception is the variable measuring the presence of the test smell *Mystery Guest*, which allowed to reject the null hypothesis Hn4 in favor of the alternative hypothesis An4. *Mystery Guest* is a test smell that appears when a test relies on external resources (e.g., files) [308]. To understand the reasons behind this finding, two of the authors jointly looked at the tests affected by this smell, trying to understand the characteristics of those tests that could justify a similar result. In the end, the two researchers come to the conclusion that there may exist an indirect relation between this smell and production

code quality. Specifically, one of the main negative consequences of having *Mystery Guest* instances is the non-deterministic behavior of the affected test code [308]. Intuitively, test classes that intermittently pass/fail cannot properly exercise the corresponding production code and find defects: thus, one likely reason behind the achieved result is the direct relation between this test smell and test flakiness [308], which therefore turns to be indirect when considering *Mystery Guest* and post-release defects. To some extent, our findings also confirm what reported by Spadini *et al.* [280] on the relation between test smells and defect-proneness of production code. This observation has to be confirmed through further empirical investigations.

The results obtained when considering the first two models (i.e., *'test'*, *'test + prod'*) also confirm the ones reported in $\mathbf{RQ}_1$; indeed, PLOC and presence of Complex Class instances are statistically significant factors only when the process variable is not taken into account.

Summing Up: The size of the test classes relates to post-release defects only if no production and process metrics are considered. We also found that *Mystery Guest* is a statistically significant factor, but a fine-grained analysis only highlighted its possible indirect relation to software quality.

10.2.3 $\boldsymbol{RQ}_3$. *The impact of dynamic test code indicators*

In the last research question, we measure how dynamically computable test code indicators are related to post-release defects. To compute these indicators, we needed to analyze tests that are executable. This restricted the scope to a dataset including 103 post-release defects. To study the effect of dynamic test code indicators, we computed and integrated them in the model coming from $\mathbf{RQ}_2$, thus building a Generalized Linear Model with 15 variables whose descriptive statistics are reported in Table 10.7. To avoid collinearity, we had to exclude PWMC, PEC, TWMC, TEC, and *Spaghetti Code*. The goodness of fit of the resulting full model was 0.225, which indicates that the model has a weak explanatory power—this because the percent by which the standard

Table 10.7: Descriptive statistics for variables used in $\mathbf{RQ}_3$. $N = 577$

Variable name	Minimum	Maximum	Mean	SD
PLOC	13.00	5077.00	259.00	342.20
isGodClass	0.00	1.00	0.12	0.38
isClassDataShouldBePrivate	0.00	1.00	0.01	0.11
isComplexClass	0.00	1.00	0.02	0.14
TLOC	5.00	2210.00	135.90	194.40
Assertion Density	0.00	0.83	0.20	0.16
isAssertionRoulette	0.00	1.00	0.86	0.34
isEagerTest	0.00	1.00	0.62	0.49
isMysteryGuest	0.00	1.00	0.06	0.23
isResourceOptimism	0.00	1.00	0.02	0.12
isIndirectTesting	0.00	1.00	0.04	0.20
Line Coverage	0.00	1.00	0.90	0.14
Branch Coverage	0.00	1.00	0.75	0.32
Mutation Coverage	0.00	1.00	0.70	0.32
pre-release changes	1.00	139.00	8.49	10.91
pre-release defects	0.00	29.00	1.37	3.66
post-release defects	0.00	23.00	0.19	1.23

deviation of the errors is less than the standard deviation of the dependent variable is pretty low: $\approx 13\%$.

Table 10.8 reports the results. When analyzing the model that only includes test-related factors, *Test LOCs* and *Mutation Coverage* are statistically significant. On the one hand, this result may indicate that larger test classes, which likely contain more tests, represent a better guard to the introduction of post-release defects. On the other hand, mutation coverage measures the ability of tests to identify artificially created defects; looking at the sign of the relation, our findings suggest that having a lower mutation coverage is related to a higher number of post-release defects, which is expected given the goal of mutation testing. Nevertheless, this variable is significant only when considering test-related factors in isolation, while its explanatory power decreases as additional factors are added to the model. In particular, the

Table 10.8: Results for $\mathbf{RQ}_3$ - The impact of dynamic test-related factors on the number of post-release defects. $N = 577$

	Test			Test + Prod.			Full		
	Estimate	**S.E.**	**Sig.**	**Estimate**	**S.E.**	**Sig.**	**Estimate**	**S.E.**	**Sig.**
Intercept	0.17	0.38		-0.16	0.37		-0.18	0.36	
Line Coverage	0.27	0.45		0.41	0.44		0.21	0.43	
Branch Coverage	-0.08	0.17		-0.08	0.17		-0.17	0.16	
Mutation Coverage	-0.55	0.18	**	-0.35	0.19	.	-0.12	0.18	
LOC (test suite)	0.00	0.00	***	-0.00	0.00		-0.00	0.00	*
Assertion Density	0.38	0.36		0.15	0.35		-0.12	0.34	
isAssertionRoulette	-0.10	0.17		-0.04	0.16		0.01	0.16	
isEagerTest	0.09	0.12		0.03	0.12		-0.04	0.11	
isMysteryGuest	-0.14	0.27		-0.02	0.27		-0.22	0.26	
isResourceOptimism	0.67	0.50		0.54	0.49		0.32	0.47	
isIndirectTesting	0.23	0.26		0.16	0.26		0.18	0.25	
LOC (production class)				0.00	0.00	***	0.00	0.00	**
isGodClass				-0.08	0.23		-0.14	0.22	
isClassDataShouldBePrivate				1.22	0.54	*	1.49	0.52	**
isComplexClass				0.26	0.45		0.37	0.43	
pre-release changes							0.03	0.01	***
pre-release defects							0.04	0.02	*
							Multiple R-squared: 0.244; Adjusted R-squared: 0.225		
							significance codes: '***'p <0.001, '**'p <0.01, '*'p <0.05, '.'p <0.1		

number of pre-release changes is among the most important factors related to software quality, while line and branch coverage do not relate to post-release defects, meaning that the amount of production code lines touched by a test does not reduce the likelihood to have faults in source code: this is in line with the recent findings of Kochhar *et al.*[145].

Line and branch coverage are not significant, regardless of the model considered. This does not allow us to reject the null hypothesis Hn5. Mutation coverage, instead, is significant only when test-related factors are considered in isolation. Also in this case we cannot reject the null hypothesis Hn6.

In this part of the dataset, with the addition of dynamic factors and process metrics, some of the previously not significant statically computable variables assumed a higher relevance. This is the case of LOC of production and test code. We extensively analyzed and discussed our data to understand the reasons behind this result and further discuss them in the following.

In $\mathbf{RQ}_2$, we considered both tests that could and could not be executed (see Figure 10.1): as such, the dataset possibly included non-executable tests having a large size which, clearly, could not exercise the production code and find defects. This might have limited the effect of TLOC on the dependent variable; conversely, when considering only executable test suites ($\mathbf{RQ}_3$), the variable turned to be significant. This statement is supported by the fact that 71% of the tests excluded from $\mathbf{RQ}_2$ have a LOC higher than the third quartile of the distribution of all test LOCs of the dataset.

A similar discussion can be done when considering the statistical significance of production code LOC. The dataset employed in $\mathbf{RQ}_3$ may have filtered out small classes exercised by non-executable tests that lead to post-release defects. To verify this hypothesis, we performed an additional analysis in which we assessed how the results of $\mathbf{RQ}_3$ change when running a model only based on statically computable test code indicators (i.e., the setting used in $\mathbf{RQ}_2$): as a result, we found that PLOC remains significant, meaning that the different statistical findings are indeed due to the specific composition of the exploited dataset. Another interesting observation concerns the relation between test and production size metrics. The directions of the two distributions (i.e., column "Estimate" in Table 10.8) are opposite, which means that: (i) the larger the TLOCs, the fewer the number of post-release defects, and (ii) the larger the PLOCs, the higher the number of post-release defects. This result is quite expected and can be better explained analyzing two relevant qualitative examples. The first one is the class `ClassDerivativeStructure` belonging to the COMMONS-MATH project; it shows a high number of PLOCs (i.e., 1,011) but at the same time a high number of TLOCs (i.e., 1,172). This class has no post-release defects, perhaps due to the robustness of the test suite. The second example is the class `BaseGenericObjectPool` in the project COMMONS-POOL; like the previous one, it is characterized by a high PLOCs (i.e., 849) but, the low number of TLOCs (i.e., 43) may have led to 23 post-release defects (the maximum number of defects among the instances we analyzed). These two examples, together with the results of the statistical model, suggest that the size of test suites can be a proxy metric to assess how robust a test is. Differently from the other research questions, the

considered variables remain significant across the four models. Indeed, also adding the process metrics, the PLOC and *'isClassDataShouldBePrivate'* are still significant.

Summing Up: Mutation coverage statistically relates to post-release defects only when test-related factors are considered in isolation. When considering both static and dynamic test code indicators, we observed production and test code size to be statistically significant in explaining post-release defects. Furthermore, our findings suggest that TLOC can be a proxy metric to assess the quality of the test.

10.3 CONCLUSION

In this chapter, we have presented an empirical study we conducted to investigate the role of test-related factors on software quality, operationalized as the number of post-release defects of production source code files. We considered eight APACHE-COMMONS systems as our case study, as they satisfy important selection criteria. We found that neither the executability nor the presence of tests is a significant factor to explain post-release defects in a statistical model. In addition, other, finer-grained test-related factors seem to have a limited effect on the number of post release defects, while we confirm the value of process factors as indicators of future software quality.

11

SOFTWARE TESTING AND ANDROID APPLICATIONS: A LARGE-SCALE EMPIRICAL STUDY

This chapter presents a large-scale empirical study on the prominence, quality, and effectiveness of the tests manually written by mobile developers. Starting from a dataset composed of 1,693 open-source ANDROID apps, we first extracted manually written test cases and computed how many and which types of tests are actually available as well as which kinds of production classes are more exercised. Secondly, we focused on the design quality of those tests, computing test code quality metrics and smells. Finally, we measured test code coverage and assertion density as proxy metrics to assess test code effectiveness. In this chapter, we provide insights into the nature of manually written tests of ANDROID apps. More specifically, we deliver the following contributions:

1. A large-scale empirical study on the prominence, design quality, and effectiveness of test cases manually written by developers in the context of mobile applications;

2. A statistical analysis into the relation of test cases to the actual defects in production code, with the aim of shedding lights into the practical usefulness of manually written tests with respect to the discovery of issues in production;

3. A qualitative investigation, in which we recruit five ANDROID testing experts and ask them to be part of a focus group [325] aimed at commenting our findings and eliciting what are the current limitations that led to them, in an effort of providing concrete insights into the new research avenues that need to be undertaken by both testing community and tool vendors to better assist developers in their daily activities;

11.1 RESEARCH QUESTIONS AND CONTEXT SELECTION

The *goal* of the empirical study is twofold: on the one hand, it aims to assess prominence, quality, and effectiveness of test cases written by mobile developers; on the other hand, it aims at identifying the key limitations of mobile testing as perceived by experts when commenting the current status of testing in mobile applications. These two goals have the *purpose* of understanding testing practices, properties, and limitations in the wild, i.e., to what extent mobile apps are tested in practice, what is the outcome of such testing, and what are the factors influencing it. The *perspective* is of both practitioners and researchers: the former are interested in observing how effective are their testing practices, while the latter are interested in understanding whether developers need new instruments to improve the quality of their test suites. In this section, we provide an overview of the research questions driving our empirical investigation and present the dataset employed to address them.

11.1.1 *Research Questions*

Our study was structured around five main research questions (**RQ**s). We started by considering some recent findings in the field of mobile software testing [41, 101, 128, 167, 196, 202], which showed that writing tests may be challenging for developers because of (i) the lack of appropriate testing tools and (ii) limited knowledge of testing practices or even willingness of developers to write tests. As such, we first analyzed the prominence of test cases in mobile applications, particularly looking at how many tests are actually developed, which types of tests are implemented and what are the kind of production classes whose functionalities tend to be exercised more. Thus, we asked:

RQ$_1$. *To what extent are test suites developed in mobile apps?*

It is worth noting that, by addressing the first research question, we also provided a larger ecological validity to some preliminary findings [62, 128] on

the extent to which mobile apps are tested. After this first analysis, we started a finer-grained investigation of test cases. First, we considered their design, as measured by test code readability and quality metrics [50, 105, 235] and test smells [195, 302, 308]. Second, we took into account the effectiveness of test cases in terms of code coverage [100] and assertion density [44, 154]. Third, we assessed the relation between test cases and post-release defects [49, 208, 209], in an effort of understanding how well can tests prevent the introduction of defects in production code. For these reasons, we defined the following research questions:

RQ$_2$. *What is the design quality of test cases developed in mobile apps?*

RQ$_3$. *What is the effectiveness of test cases developed in mobile apps?*

RQ$_4$. *What is the relation between test cases and post-release defects in mobile apps?*

The analyses of these research questions allowed us to provide a detailed overview of the extent, quality, effectiveness, and fault detection capabilities of tests available in mobile applications. Such an overview was then presented to testing experts with the goal of assessing the quantitative findings against their opinion/expertise and identifying the major reasons behind the current state of testing in mobile applications. This led to our final research question:

RQ$_5$. *What is the developer's take on the current state of mobile apps testing?*

By definition, our methodology followed a mixed-method research approach [61] where the insights derived by mining mobile app data are complemented with qualitative cues coming from experts working on mobile app testing on a daily basis and that can contribute to the development of a comprehensive state of the practice on the matter.

11.1.2 *Context of the Study*

The *context* of the empirical study consisted of mobile application data (**RQ**$_1$-**RQ**$_4$) and testing experts (**RQ**$_5$).

As for the former, we considered a set of 1,693 open-source ANDROID apps gathered by mining F-DROID,[1] a repository of free and open-source mobile applications that has been widely employed in the past [53, 134, 198, 237, 267] and that contains a set of applications that enables a good generalizability of the findings with respect to the overall population of free and open-source mobile apps [70, 146, 153, 267]. It is important to note that, while F-DROID contains over 3,000 apps, we narrowed our selection in order to only consider repositories on GITHUB;[2] furthermore, we manually excluded duplicated apps and forks of those already existing in the repository. Based on these filters, we ended up with the final 1,693 open-source mobile apps. Our analyses have been conducted on test cases written in Java. We are aware that this is not the only language available to develop tests in mobile applications but it is certainly among the most diffused. In order to reinforce the validity of our study, we sift through our dataset in order to count the number of tests written in Kotlin, a language that has continued to gain adoption in mobile apps testing over the last year [190]. As a result, we found that only 139 out of the 1,693 applications ($< 1\%$) contain at least one Kotlin test. Even if a few applications contain a reasonably high number of Kotlin tests ($\approx$100), we decided not to conduct further analyses, since these just represent rare and isolated cases over the considered dataset.

As for the testing experts, we recruited five professional mobile developers with an ANDROID programming experience ranging between 5 and 10 years. They all work in industry and, on average, they have been developing or contributing as testers to the creation of 25 apps each. While all the participants currently work in industry, two of them still contribute to open-source ANDROID applications, while the other three have worked on open-source applications in the past. The size of the population of developers considered was driven by the specific research approach employed to address **RQ**$_5$: focus

1 https://f-droid.org
2 https://github.com

groups are a form of qualitative research that involves a small number of people sampled conveniently and that can provide expert judgments on the subject of interest [325]. According to well-established guidelines, the ideal size of a focus group is five to eight participants [55]: indeed, larger focus groups are difficult to control and, more importantly, limit each participant's opportunity to share insights and observations [55]. More details on the selection of this research approach as well as on the methodology employed to recruit participants are reported in Section 11.6.

11.2 RQ$_1$ - ON THE PROMINENCE OF TEST CASES IN MOBILE APPS

This section discusses the research methodology and the results achieved when investigating the prominence of test cases in the considered set of mobile apps.

11.2.1 *Research Methodology*

To address **RQ**$_1$, we first quantified the number of test classes available for each of the apps in our dataset. Starting from their GITHUB repositories, we cloned the apps locally and, afterwards, we performed an exhaustive search through their packages in order to extract classes having *"Test"* as prefix or suffix. As a result of this search process, we computed the number of test classes and methods per app, which corresponds to the number of test suites and test cases available in a mobile application. Furthermore, we proceeded with a more detailed analysis of test suites that aimed at classifying them according to their granularity (e.g., unit vs. integration) and type (e.g., performance). As an automatic classification was not possible, we manually analyzed all the 5,292 extracted test suites using a grounded theory-based methodology [255] which involved two of the authors (from now on, the *inspectors*). It is worth noting that, in order to exclude false positive tests, during this manual investigation we also checked if the considered classes were actually test classes. No false positives came out from this analysis.

The process consisted of two steps:

Tuning phase. Initially, the inspectors independently classified the same set composed of 500 test suites and annotated in a spreadsheet their granularity and type(s). Whenever possible, the inspectors relied on the available documentation (e.g., code comments) to understand the properties of a certain test: for instance, if developers explicitly stated that the test suite covered the corresponding production class, then the inspectors marked it as a unit test. In the other cases, the inspectors relied on the name of the class as well as analyzed its content to check if (i) only a production class was exercised, i.e., it was a unit test, (ii) more classes were involved to verify the interactions among components, i.e., it was an integration test, or (iii) otherwise, it was a system test. A similar strategy was employed when classifying the type: whenever possible, the inspectors relied on the documentation, while in other cases they manually went over the code to understand which functional or non-functional requirement was exercised. Particularly hard was the case of energy-related tests, for which the inspectors verified whether the test code contained any identifier, API of a third-party library, or profilers of the ANDROID platform connected to energy management. To provide the reader with a concrete example of the classification made, let consider the case of the `ProgramMemoryTest` class of the FINNEYPOKER app. This test suite aims at assessing the memory consumed by the animations implemented in the `Animator` class, which is used by the `PokerActivity`, i.e., the main UI class of the app. As such, the (i) granularity of the test suite was categorized as *'integration'*, since it did not involve one class in isolation nor the system as a whole, and (ii) the type was associated to *'performance'*, as the goal was to assess the consumption of the app in certain conditions.

Through the classification of the same test suites, the inspectors could tune their judgments, find a common way to classify granularity and type of the considered test suites, and discuss their disagreements to better understand the reasoning done by the other inspector. Furthermore, they could compute an initial coding agreement using the Krippendorff's alpha Kr_α [152]. This measured to 0.92, that is considerably higher than the 0.80 standard reference score [11] for Kr_α.

Classification phase. Once completed the tuning phase, the inspectors classified the remaining 4,795 test suites, by analyzing 2,397 and 2,398 each. The outcome allowed the creation of a test suite granularity and type taxonomy for ANDROID apps, which we discussed in Section 11.2.2.

As an additional analysis aiming at addressing our first research question, we quantified how many and which types of production classes are tested. In this way, we could understand whether developers tend to test only certain specific types of classes (e.g., `Activity` or `Fragment` classes) as well as how much of the production code is covered by a test suite.

To enable this analysis, we first needed to link production to test classes. We relied on the pattern-matching approach designed by Van Rompaey and Demeyer [310]: for each test class, it removes the string *"Test"* from its name and search the production class that matches the remaining part of the name. For instance, using this strategy the test suite *MainActivityTest* would be linked to the production class named *MainActivity*. It is worth mentioning that this linking approach is lightweight in nature and can scale up to the number of apps considered in our study; yet, it has shown similar performance with respect to more sophisticated test-to-code traceability techniques [310].

Afterwards, we computed the number of production classes having a corresponding test suite. As for the type of production classes tested, we performed a first automatic classification, based on keywords, and then we double-checked the classification manually. Specifically, we defined a set of keywords that can distinguish GUI, application logic, and storage components of an ANDROID app. For instance, the GUI keywords included *"activity"* and *"fragment"*, which generally characterize `Activity` and `Fragment` classes used by developers to develop the graphical interface of the app. Since this automatic classification may be erroneous in some cases, one of the authors double-checked it and corrected the labels assigned whenever required.

11.2.2 *Analysis of the Results*

Table 11.1 reports descriptive statistics on the number of test suites and test cases available in the considered dataset as well as the overall number of

Table 11.1: Descriptive Statistics of the mobile apps analyzed.

	#Test Suites	#Test Cases
Min	0	0
Max	205	2045
Average	3.24	63.30
Median	0	5
Standard Deviation	14.07	202.80
	% Apps Tested	**% Apps Not Tested**
	40	60

mobile applications containing at least one test class. As shown, the first thing that leaps to the eye is that 60% of apps do not present any test case: as such, we can confirm the results obtained by previous work which proved that mobile apps, and in particular ANDROID ones, generally lack tests [62, 146, 274]. This finding reinforces the need for further research on the topic of mobile app testing and, specifically, how to convince developers—who may be non-experienced with the development of source code [319]—of the importance of testing their apps, e.g., by means of empirical evidence showing how lack of testing may worsen the quality of mobile apps. For instance, our results motivate and promote investigations aimed at relating test code quality to change/fault-proneness of the apps [22, 267] or the commercial success of mobile applications [40, 233].

Narrowing our attention to the applications that are actually tested, i.e., the 40% of the apps in our dataset, we computed descriptive statistics related to both test suites and test cases. Table 11.1 reports the results of this analysis. Looking at the minimum and maximum number of test cases, we found a high variability among the considered applications: indeed, the minimum size of test suites is zero, while it reaches 205 in the best case, with a mean of about three test classes. This result clearly highlights that even apps having Java test suites are in general poorly exercised and would need further support in this activity. The standard deviation value (202.80) confirms the high variability among the considered apps.

As a second part of our analysis aimed at addressing $\mathbf{RQ}_1$, we classified test suites according to their granularity and type. Table 11.2 summarizes our

Table 11.2: Granularity and type of test suites developed in the dataset.

Granularity		
Name	Abs.	Rel.
Unit	3,872	73%
Integration	1,273	24%
System	147	3%
Type		
Name	Abs.	Rel.
Functional	4,619	87%
Performance	190	4%
Energy	145	3%
Portability	133	3%
Security	104	2%
Usability	101	1%

results. In the first place, we can notice that most of the test suites analyzed are at unit-level: 73% of the tests in our dataset are indeed at this granularity. Interestingly, we discovered that 3,605 of them are directly related to a single production class, while the remaining 268 unit tests exercise more classes at the time. For instance, tests named `IntentTest` or `SwipeTest` indicate generic tests that exercise common functionalities of certain classes without focusing on some of them specifically.

Furthermore, we found that 24% of the test suites pertain to integration testing and aim at exercising how components behave when working together. Finally, a small portion of the considered tests (3%) consists of system tests that aim at testing the application as a whole. Perhaps more interestingly, our investigation into the types of test classes written by developers revealed the existence of a taxonomy composed of six types. As expected, most of the test suites refer to functional tests (87%), namely tests that exercise the input/output of production code classes: this confirms the findings of previous researchers who found that functional testing is the most widely spread type of testing [28, 45]. Subsequently, our categorization shows that performance tests represent the 4% of the available tests: while this number is way lower

than the functional tests, this seems to indicate that (1) developers care, even if in a lower extent, of performance of mobile apps, thus confirming previous findings in the field [67, 140] and (2) performance testing is a more delicate problem than for traditional applications [156, 205], suggesting the need of more research to understand better why this happens and what are the consequences.

Furthermore, we found the energy testing is the third more popular type of exercising mobile apps. Also in this case, the number of tests is substantially lower than the one of functional tests; these results are in line with previous findings that highlighted that more automated support to this type of testing would allow developers to better exercise the energy aspects of mobile apps [200, 231]. A small percentage of test suites in our dataset relates to portability testing, namely the types of tests that verify whether the functionalities of an application is compatible with previous versions of the ANDROID operating system [321]: the small amount of tests in this category suggests that more research would be needed in order to understand the reason behind these achievements [194]. Finally, security and usability testing represent the least prominent types of tests in the exploited dataset. On the one hand, the very small amount of tests in these categories clearly highlight that developers are not properly aware of how to cover these aspects [93, 99, 254]: this is particularly worrying in the case of security, also considering the recent data provided by NOWSECURE[3], which showed that (i) 35% of communications sent by mobile devises are un-encrypted, (ii) 25% of apps have high-risk security flaws, e.g., expose private or sensitive data about a user or their activity, and (iii) 82% of ANDROID devices use an outdated version of the operating system. On the other hand, our findings support and motivate the research done on usability and GUI testing [101, 182], which has been an active field over the last years.

Finally, we focused on the production classes that are actually exercised. Table 11.3 reports the results achieved: specifically, we split the classes based on their role in the system and, according to this classification, we identified three main categories, namely, *GUI*, *Storage*, and *Application Logic*: the first

3 A well-known security company targeting mobile apps: https://tinyurl.com/rdhrszc

Table 11.3: Types of production classes tested. Abs. = Absolute number; Rel. = Relative number.

Activity		Intent		Fragments		Storage		Application Logic	
Abs.	Rel.	Abs.	Rel.	Abs.	Rel.	Abs.	Rel.	Abs.	Rel.
202	6%	9	1%	52	1%	114	3%	3,228	89%

Table 11.4: Percentage of tested classes per production class type.

Type	# tests	# production_classes	tests/production_classes %
Activity	202	8,202	2%
Intent	9	38	24%
Fragments	52	5,362	1%
Storage	114	1,713	7%
Application Logic	3,228	14,109	23%

refers to production classes implementing the logic behind the graphical user interface of mobile apps, the second to the classes that manage the storage of the apps, while the latter to the classes having the single responsibility of implementing business logic of the apps. For the sake of comprehensibility, we split the *GUI* category in the three main class types, namely `Activity`, `Intent`, and `Fragment`. These are the class types that ANDROID developers use to develop the user interface of their applications.

Looking at Table 11.4, we can observe that most tests in the dataset exercise the application logic of mobile apps. Behind this result, there might be different explanations: First, developers are not properly supported nor aware of current techniques when it comes to the testing of other aspects of their apps [62]. Second, mobile developers are sometimes junior or with less experience than programmers working in other domains and, as shown by previous researchers, they might be less aware of the importance of testing, hence limiting themselves to exercise a limited amount of classes [254].

Summing Up: Mobile applications contain very few java tests, indeed, only 40% of the apps contain at least one test suite. As for the tested apps, most of the tests pertain to unit tests that exercise the functionalities of the app, while other aspects are not widely considered, like for instance, performance of GUI testing.

11.3 $\mathbf{RQ}_2$ - ON THE DESIGN QUALITY OF TEST CASES IN MOBILE APPS

This section reports methodology and results of our analyses to address $\mathbf{RQ}_2$.

11.3.1 *Research Methodology*

Given our original dataset, we had to exclude all the apps without tests from this second research question. This process led us to focus on 673 mobile apps. To assess the design of the considered test cases we covered three macro-aspects that can characterize their maintainability and understandability. Table 11.5 summarizes the metrics adopted to address $\mathbf{RQ}_2$. The selection of these metrics was based on the findings reported by Grano *et al.*[102], which presented a taxonomy of metrics deemed significant by developers to measure test code quality. More specifically, such a taxonomy includes behavioral, structural, and execution high-level concerns that developers consider relevant when developing tests and that can be mostly quantified using the metrics in Table 11.5.

In the first place, we considered test code quality metrics, relying on the metric suite originally defined by Chidamber and Kemerer [50] and other metrics related to code quality. According to Grano *et al.*[102], this set of metrics addresses or helps addressing aspects like scope, size, reusability, and independence of test cases. We computed the Lines of Code (LOC): according to previous achievements [149, 242, 337], having higher size may cause issues for developers with respect to the maintainability of tests as well as to their fault-proneness [280, 306]. For similar reasons, we computed

Table 11.5: List of factors considered in order to measure the design quality of test cases.

Group	Name	Description
Code Metrics	LOC	Number of lines of code of the Test Class
	WMC	Weighted Method Count of the Test Class.
	RFC	Response for a Class.
	IFC	Information Flow Coupling.
	LCOM5	Lack of Cohesion of Test Methods.
	TCC	Tight Class Cohesion.
	LCC	Loose Class Cohesion.
Textual Metrics	Readability	The readability level of the test.
	Comment ratio	Ratio between lines of comments and lines of source code.
Test smells	Eager Test	A test exercising more methods of the production target.
	Indirect Testing	A test interacting with the target via another object.
	Resource Optimism	A test that makes optimistic assumptions on the existence of external resources.
	Mystery Guest	A test that uses external resources (e.g., files or databases).
	Assertion Roulette	A test method containing several assertions with no explanation.

cohesion metrics such as Lack of Cohesion of Test Methods (LCOM5 [122]), Tight Class Cohesion (TCC), and Loose Class Cohesion (LCC) [60]; we measured different metrics as they can provide orthogonal information that may be useful to analyze the cohesion of tests better [305]. Furthermore, we considered the coupling between tests, which is one of the most critical problems when comprehending test code [102, 335]. To this aim, we computed the Information Flow Coupling (IFC), a metric that captures the relations between tests in terms of information exchanged [305] and is among the best suited for assessing the quality of tests [91]. Finally, we considered the complexity of test code. In this case, the rationale comes from previous studies [68, 102, 210, 341] which showed that complexity metrics may be

related to both scope and defectiveness of test code as well as may lower the overall understandability of the target of tests [335]. We quantified complexity by computing Weighted Methods per Class (WMC) and Response for a Class (RFC): the former represents the sum of the complexity of the test cases included in a suite, while the latter estimates complexity by considering the number of methods that can potentially be executed in response to a message received by an object of a class. All the metrics were computed at test suite-level, as they can be only extracted at this granularity.

The second set of metrics relate to textual aspects of source code. These can be helpful when quantifying the readability and diagnosability (i.e., the ability of developers to understand faults detected by a test) of test code [102]. First, we computed the overall readability of test cases by relying on the automated approach proposed by Buse and Weimer [36] - we employed the original tool proposed by the authors. Such an approach employs a machine learning-based solution that internally computes 19 metrics covering various aspects of source code that may influence its readability, like the number of keywords or the number of spaces in a piece of code to name a few. The output of the readability tool consists of a readability index ranging between 0 and 1, where 0 indicates an unreadable code and 1 a perfect readability. We also computed the comment ratio, namely the percentage of comments per test method lines of code - the higher this ratio the higher the documentation available for developers and, therefore, the higher its understandability.

To complement the analysis of test code quality metric profiles, we considered test smells, i.e., poor design or implementation choices applied by programmers during the development of test cases [308]. On the one hand, test smells make test code more change- and fault-prone [280] as well as harder to comprehend and maintain [25]. On the other hand, test smells have been shown to be one of the primary causes behind test instability, thus making them extremely harmful for developers [74]. We focused on five forms of test smells widely investigated by the research community, namely *Mystery Guest*, *Resource Optimism*, *Eager Test*, *Assertion Roulette*, and *Indirect Testing*. Their definitions are provided in Table 11.5. To detect them, we employed the code metrics-based tool developed by Bavota *et al.*[25], which has shown

Table 11.6: Descriptive statistics for all metrics considered in RQ_2. Outliers have been removed from distributions.

Metric	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
LOC	2.00	14.00	32.00	46.40	66.00	181.00
WMC	0.00	2.00	4.00	4.80	7.00	17.00
RFC	0.00	6.00	17.00	26.30	39.00	112.00
IFC	0.00	0.19	0.36	0.37	0.53	1.00
LCOM	0.00	0.27	0.50	0.50	0.75	1.00
TCC	0.00	0.00	0.00	0.26	0.50	1.00
LCC	0.00	0.00	0.50	0.50	1.00	1.00
Readability	0.00	0.00	0.00	0.13	0.01	1.00
Comment ratio	0.00	0.00	0.00	0.04	0.03	0.40

to have high accuracy, close to 86% of F-Measure [25, 241] and has been validated several times in previous work [95, 230, 235, 280], thus making us confident of its suitability for our study. The detector identifies test smells at class-level granularity, which is the same as the other considered metrics. In particular, for each test suite and test smell type, the detector provides a boolean value reporting whether the suite contains at least one instance of the test smell type under investigation.

11.3.2 *Analysis of the Results*

Table 11.6 reports the distributions for all the quality metrics considered in our second research question—note that outliers have been removed from the table for the sake of comprehensibility.

Looking at the table, we first noticed that the LOC metric, which computes the size of test suites, has a median value of 32.00, meaning that the vast majority of the considered tests have a limited size. There are, however, several outliers: we manually analyzed them to better understand how are they composed. From this analysis, we found that all the outliers refer to apps having only one big test class containing several test methods that

exercise production code belonging to different classes. As an example, the test `MainActivityTest`, belonging to the package `opencamera.test` of the OpenCamera app, has 12,637 lines of code and implements 1,188 test methods.

When considering complexity metrics like WMC and RFC, our findings suggest that the complexity of tests is generally low (median of 4.00 and 17.00 for WMC and RFC, respectively). The discussion for coupling is more interesting: indeed, the IFC metric has a median of 0.36: this indicates that there exist a non-negligible number of test suites containing methods that depend on other methods of the same class. Besides making such tests less comprehensible [335], this phenomenon may potentially lead to undesired issues like, for instance, potential flakiness due to a test ordering problem, which arises when the execution of a test depends on the execution of another one [172].

Turning the attention to the test case cohesion, we can provide a number of observations. First, the LCOM is almost equally distributed over the spectrum of possible values for this metric. Given its definition, this result indicates that there is a fairly similar amount of test cases that use and not use instance variables defined in the test suite; from a practical perspective, this possibly indicates that the design of tests and their inter-dependence may be affected by the way specific developers implement test cases (e.g., their experience or knowledge of the domain [44, 254]).

The analysis of TCC and LCC provided us with further insights into the cohesion of tests. While the former measures the number of test pairs that directly share instance variables of the test suite, the latter indicates how many of them are either directly or indirectly connected (i.e., share the same variables or are directly connected to the same test). The distribution of those metrics tell us that, while test methods are not always directly connected, they have more often an indirect connection. As such, they follow a similar trend as the one shown in previous studies done on the code quality profile of production classes [75]—there seems to be no peculiarities of these metrics that distinguish tests written by mobile developers.

Finally, we could observe that both the textual metrics considered have a median equals to 0, with a third quartile of 0.01 and 0.03 for readability and comment ratio, respectively. On the one hand, these results suggest that mobile developers spend almost no time addressing documentation concerns in test cases: the low distribution of values for the comment ratio, indeed, reports that only in a few cases developers add comments that explain the responsibilities of the test. On the other hand, the analysis of readability is somewhat even more worrisome: not only tests are not documented, but also potentially hard to comprehend for developers - note that previous literature has shown that poor test readability is often connected to a decrease of bug detection capabilities [105, 333].

Table 11.7: Absolute and relative number of test smells detected. AR = Assertion Roulette; ET = Eager Test; MG = Mystery Guest; RO = Resource Optimism; IT = Indirect Testing.

AR		ET		MG		RO		IT	
Abs.	**Rel.**	**Abs.**	**Rel.**	**Abs.**	**Rel.**	**Abs.**	**Rel.**	**Abs.**	**Rel.**
2,508	50%	1,556	31%	439	9%	123	3%	371	7%

As for the test smells, Table 11.7 reports the distribution of design issues over the considered set of mobile apps. In the first place, we can confirm previous findings in the field [25, 26, 103] and claim that test smells have a high diffuseness also when considering the mobile context. Most of the instances found (50%) refer to *Assertion Roulette*, namely the smell that arises when there are multiple `assert` statements without explanation—this smell lowers understandability and maintainability of test suites [308]. Instances of *Eager Test* are also quite diffused and affect 31% of the test suites in our dataset. According to previous results [280], this smell type is associated with a lower effectiveness of the affected test in terms of fault detection capabilities. The other test smells are less diffused: *Mystery Guest* appears in 9% of the considered tests, while *Resource Optimism* and *Indirect Testing* in 3% and 7% of the cases, respectively. These percentages are in line with those found in traditional systems by Bavota *et al.*[26] and Grano *et al.*[103],

thus indicating that test smells have similar diffusion and relevance in both contexts.

More in general, from our empirical analyses we observed that, while the structural metric profile of tests would not show potential problems affecting their design, the quality of tests is still threatened by the presence of test smells [280]. Despite the fact that they capture two different concepts, this contradiction may potentially indicate that currently available metrics are not enough to measure the actual quality of test suites and, as such, new, different test code metrics that better capture the design quality and understandability of test suites should be further studied and defined.

Summing Up: The metric profile of the considered test suites does not always indicate the presence of possible issues in test code. However, tests are often affected by test smells that may possibly negatively influence their effectiveness, for instance by leading them to miss faults in production code. Our findings suggest the need for new test code metrics that can better measure the actual quality of test suites.

11.4 RQ$_3$ - ON THE EFFECTIVENESS OF TEST CASES IN MOBILE APPS

This section details the methodological steps conducted to address our third research question and the results achieved.

11.4.1 *Research Methodology*

Test code effectiveness can be estimated in different ways. In the context of our study, we focused on two complementary aspects that have been shown to influence the ability of tests to catch defects in production code, namely line coverage [322] and assertion density [154]. The former measures the amount of code that has been exercised based on the number of Java byte code instructions called by the tests: from a practical perspective, we employed the JaCoCo Android plug-in,[4] a popular code coverage tool, to compute the

4 https://github.com/arturdm/jacoco-android-gradle-plugin

value for each of the considered test suites. As for the assertion density, this is defined as follow:

$$assertion.density(tc) = \frac{\#assertions(tc)}{LOC(tc)} \tag{11.1}$$

where tc is the test case under consideration, $\#assertions(tc)$ is the number of assert statements in tc and $LOC(tc)$ is the number of lines of code of the test. Note that we employed the definition of assertion density introduced by Kudrjavets *et al.*[154]. We considered this metrics since it has been associated in the past with a reduction of defect density in production code [44, 154], hence providing an indication of how good a test suite can actually be. To compute assertion density, we developed our own tool.

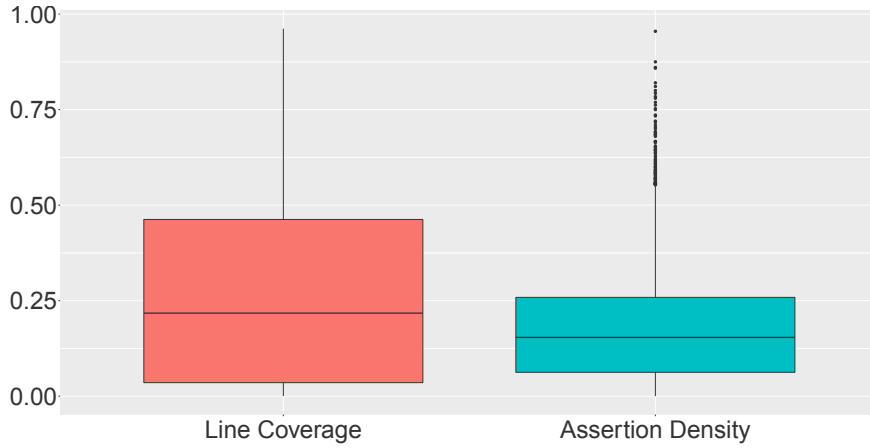

Figure 11.1: Distribution of test code quality metrics in our dataset.

11.4.2 *Analysis of the Results*

Figure 11.1 reports the distribution of line coverage and assertion density among all the applications of the dataset. As the figure shows, the values of both the metrics are between 0 and 0.5, excepting for some outliers. The median for line coverage is equal to 0.23, while the one of assertion density

is 0.17. We also observe a notable number of outliers, especially when considering the assertion density. Nonetheless, we can claim that these values relate to low effectiveness [185]: their effect on the post-release defects of mobile applications is investigated in the context of $\mathbf{RQ}_4$.

When considering line coverage, the discussion is similar. The vast majority of the test suites have low coverage and cannot properly exercise the corresponding production code. Unlike assertion density, for line coverage we noticed something peculiar and worth of discussion: in some cases, developers discuss about code coverage on the issue trackers and, particularly, on the way they can increase it. For instance, let consider the AnySoftKeyboard app: the developers in this case adopt a pull-based development process where all changes must pass through a pull request before being merged. In most of the cases where new code is committed, developers explicitly ask to the author of the change to verify that the code coverage of unit tests is high enough. As an example, in the issue #551, one developer applied multiple changes to the test code in order to increase its coverage up to 87%. We found similar cases when considering other applications, thus leading us to claim that the developer's perception of code coverage is sometimes pretty high and reflected into the way test cases are developed—this result partially contradicts what reported by Linares-Vásquez *et al.*[165] through their study on the developer's perception of code coverage and indicates that further experiments would be desirable to understand the real value of code coverage for developers. Nevertheless, our findings suggest that mobile programmers still experience troubles when developing effective tests.

Summing Up: The median code coverage and assertion density are 0.23 and 0.17, respectively. The effect of these low values is analyzed in the next research question. Furthermore, we found that in some cases developers perceive code coverage as highly relevant to accept pull requests.

11.5 $\mathbf{RQ}_4$ - ON THE RELATION OF TEST CASES TO POST-RELEASE DEFECTS IN MOBILE APPS

In this section, we describe methodology and results pertaining to $\mathbf{RQ}_4$, i.e., the relation between test cases and post-release defects of mobile apps.

11.5.1 *Research Methodology*

In the context of this research question, we adopted a statistical approach with the aim of relating and assessing how test-related metrics characterizing the goodness of the manually written test cases in mobile applications can indicate the statistical likelihood to have defects in production code. In more practical terms, while in $\mathbf{RQ}_3$ we measured the effectiveness of test cases only based on metrics computable considering the test code itself, in $\mathbf{RQ}_4$ we assessed whether and how the goodness of test cases is reflected to the quality of production code, as measured by the number of post-release defects.

The statistical approach employed consisted of multiple steps and methodological choices. The remainder of this section explains our approach in terms of model dependent and independent variables as well as of methods applied to enable valid statistical conclusions and interpretations.

Dependent variable. The dependent variable considered in $\mathbf{RQ}_4$ is the number of post-release defects, namely the amount of bugs affecting the mobile applications in our study after the snapshot considered for the analysis. The GITHUB repositories pertaining to those apps offer the entire change history in form of commits. We first determined if a commit fixed a defect. In this regard, we searched for issue IDs in commit messages by finding matches with the prefix used in the bug tracker system. Once retrieved a commit referencing an issue, we queried the mobile app's issue tracker system in order to filter only issues related to resolved bugs. Then, we relied on the keyword-matching technique devised by Fischer *et al.*[80] to analyze the commit message and search for the presence of specific keywords, e.g., *'bug'*, *'fix'*, or *'defect'*, that are typically used by developers to mark defect-fixing activities. The application of this technique allowed us to filter out all

those commits that referred to issues in the issue tracker but were related to other maintenance and evolution activities. Despite the technique might be considered naive, empirical assessments have shown an accuracy of ≈80% [80, 236] and, for this reason, often been used by previous research [130, 141].

After detecting all defect-fixing commits, we applied the Śliwerski-Zimmerman-Zeller (SZZ) algorithm [278], which is able to identify the defect-inducing commits, namely those modifications that likely introduced defects. The algorithm relies on basic `Git` features such as annotation and blame. Given a defect-fixing commit k as input, SZZ works as follows:

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

- Starting from the revision *rel-fix*$_{i,k} - 1$, for each source line in f_i changed to fix the defect k, SZZ identifies the production class C_j to which the changed line belongs. Furthermore, the `blame` feature of `Git` is used to identify the revision where the last change to that line occurred. In so doing, blank lines and lines that only contain comments are identified and excluded using an island grammar parser [199]. This produces, for each production class C_j, a set of $n_{i,k}$ defect-inducing revisions *rel-defect*$_{i,j,k}$, $j = 1 \dots n_{i,k}$. As such, more than one commit can be marked by SZZ as defect-inducing.

Once extracted the defect-inducing commits, we finally computed the post-release defects of a production class as the number of defect-inducing activities involving the class after the release date of the snapshot of the mobile apps considered. From a technical perspective, we employed the SZZ algorithm implemented within PyDriller[5] to compute the dependent variable.

5 Link: https://pydriller.readthedocs.io/.

Independent variables. Our aim was to verify the extent to which the goodness of test cases implemented by mobile developers relate to post-release defects. As such, the independent variables of the statistical model comprise the set of metrics considered in **RQ**$_2$ and **RQ**$_3$. This way we could analyze the impact of various features, i.e., static and dynamic factors, textual metrics as well as test code design, on the ability of tests to act as a guard for the introduction of defects in forthcoming versions of mobile apps.

Confounding factors. Other than to test-related features, the number of post-release defects might be due to additional aspects pertaining to production code quality or the development process. As an example, larger production code classes might be more defect-prone independently from the test cases exercising it. To account for this aspect and avoid a biased interpretation of the results, we computed a set of confounding features that have been shown to influence the defect-proneness of source code. These are summarized in Table 11.8. As shown, they cover four main characteristics of product and process quality:

- Among the structural metrics, we first take the Production Lines Of Code (PLOC) metric into account. It measures the size of the production classes. Its selection was driven by the fact that PLOC has been often associated to an increase of fault-proneness [149, 242, 337]. To compute it, we employed the automated tool developed by Spinellis [282].

- The Weighted Method per Class (PWMC) and Response for a Class (PRFC) metrics [50] measure the complexity of production code, which is something that naturally influences the defect-proneness of source code [68, 210]. The tool by Spinellis [282] is able to compute this metric too.

- Coupling is another aspect strongly connected to software quality [91]. In our work, we computed the Information Flow Coupling (PIFC) of production classes, i.e., a metric describing the relation between classes in terms of information exchanged.

Table 11.8: List of confounding factors used in the study.

Group	Name	Description
Structural metrics	PLOC	Number of lines of code of the production class.
	PWMC	Weighted Method Count of the production class.
	PRFC	Response for a production class.
	PIFC	Information Flow Coupling of the production class.
	PLCOM5	Lack of Cohesion of Methods of the production class.
	PTCC	Tight Class Cohesion of the production class.
	PLCC	Lose Class Cohesion of the production class.
Code smells	God Class	A class having a large size, poor cohesion, and several dependencies with other data classes of the system.
	Class Data Should Be Private	A class exposing its attributes, thus violating the information hiding principle.
	Complex Class	A class presenting a overly high cyclomatic complexity.
	Functional Decomposition	A class implemented as a function.
	Spaghetti Code	A class that exhibit a functional-style programming structure, declaring a number of long methods without parameters.
Android-specific smells	Durable Wakelock	A class acquiring a wake-lock without releasing it.
	Inefficient Data Structure	A class that declares a HashMap local variable whose first type argument (i.e., key) is an Integer.
	Internal Setter	A class containing one (or more) non-static method(s) that calls a setter having only a single assignment.
	Leaking Thread	A class exhibiting a Thread that is started but not interrupted.
	Member Ignoring Method	A class containing a non-static and non-empty method that (i) does not access any instance variable; (ii) does not use this and super keywords; (iii) does not override an inherited method.
Development process	Pre-release changes	Number of changes involving the production class before the release date of the considered snapshot.

- Cohesion has also been associated to fault proneness in the past [19]. In this respect, we measured cohesion of production classes by mean of Lack of Cohesion of Methods (PLCOM5), Tight Class Cohesion (PTCC), and Lose Class Cohesion (PLCC).

- Code smells are indicators of sub-optimal design/implementation choices in source code [88]. A number of previous papers have established that those smells heavily increase the chance of production code being faulty [64, 116, 137, 225, 226, 243, 304]. On the one hand, we considered five traditional code smells from the catalog by Fowler [88]. These have different characteristics and cover various program entities, i.e., *God Class*, *Class Data Should Be Private*, *Complex Class*, *Functional Decomposition*, and *Spaghetti Code*. On the other hand, we took the peculiarities of mobile applications into account and computed the so-called Android-specific code smells, i.e., design flaws that are specific of mobile apps and that can impact on defect-proneness in different manners, e.g., by increasing the chance of functional and energy-related defects. In this regard, we computed the 5 code smells mentioned in Table 11.8.

  As for the actual detection of these code smells, we relied on DECOR [197] for the traditional ones and on ADOCTOR [124] for the Android-specific ones. Both tools have been extensively validated by the research community and showed an excellent accuracy [124, 228], hence representing valid tools to use for our purposes.

- Finally, we computed the number of pre-release changes. This metric captures the quality of the development process [119] and can highlight relevant complementary evolutionary aspects. The metric was computed by mining the change log of the considered apps and counting how often a certain production class has been modified.

Statistical approach. As last step to address our research question, we built a statistical model relating independent and confounding metrics to post-release defects. We opted for the construction of a Generalized Linear Model

(GLM) [213]: it models the relationship between a scalar response, like the number of post-release defects, and one or more explanatory variables, i.e., the set of independent and confounding factors, by fitting a linear function whose unknown model parameters are estimated from the data; we used the *'Gaussian'* family when implementing the model. The reason behind the choice of this statistical approach was twofold. At first, it simultaneously analyzes the effects of both confounding and independent variables on the response variable [112]. Secondly, it does not require the normality of data distribution: in our case, the Shapiro-Wilk normality test [270] rejected the null-hypothesis, hence indicating that our data is not normally distributed.

To properly interpreting the statistical results, we accounted for possible issues with multicollinearity [217]. In doing so, we run a hierarchical clustering based on the Spearman's rank coefficient [281] to cluster together variables at different levels of correlation. Afterwards, if two of them had a correlation higher than 0.6, we excluded the more complex one from the model.

Finally, we interpreted the output of GLM by analyzing the statistical significant codes it assigns to each explanatory variable: if a certain metric is statistically significant, this implies that the chances of the effect on the number of post-release defects being random is sufficiently low. We also computed the Adjusted R-squared [73] to assess the goodness of fit of the model, a metric indicating how close the data is to the fitted regression line.

11.5.2 *Analysis of the Results*

Table 11.9 reports the statistical results obtained. Before discussing them, it is worth pointing out that from the total set of variables employed within the model, we had to exclude (i) the variable signaling the presence of the *Functional Decomposition* code smell because it was too correlated with p_lcc, (ii) p_wmc due to its high correlation with p_loc, and (iii) t_wmc and t_rfc, which in this case had high correlation with t_loc. In addition, the statistical model could not consider the *Member Ignoring Method*, *Inefficient Data Structure*, and *Leaking Thread* smells because the Android-specific

Table 11.9: Results for **RQ**$_4$ - Factors influencing post-release defects in mobile apps.

	Estimate	S.E.	Sig.
Intercept	-1.18	1.14	
TLOC	-0.01	0.01	
TIFC	1.67	1.11	
TLCC	0.04	0.67	
TTCC	1.91	0.56	***
Assertion Roulette	-0.23	0.53	
Eager Test	-0.34	0.46	
Mystery Guest	1.03	0.81	
Resource Optimism	-2.73	1.37	*
Indirect Testing	1.29	0.63	*
Readability	3.74	2.30	
Comment Ratio	-15.43	7.07	*
Line Coverage	0.31	0.94	
Assertion Density	1.48	1.61	
PLOC	-0.06	0.02	**
PIFC	0.60	1.00	
PLCC	1.94	1.19	
PRFC	0.33	0.99	
PTCC	-0.51	0.71	
God Class	2.51	1.32	.
Class Data Should Be Private	6.91	5.25	
Complex Class	11.49	4.32	**
Durable Wakelock	1.00	4.30	
Internal Setter	2.84	3.51	
Pre-release Changes	0.78	0.05	***
Multiple R-squared: 0.169; Adjusted R-squared: 0.114			
significance codes: '***'p <0.001, '**'p <0.01, '*'p <0.05, '.'p <0.1			

code smell detector, i.e., aDoctor, did not detect any instance for these smells.

Overall, the model was composed of a total of 24 metrics. The Adjusted R-squared measured 0.114: the value is pretty low, meaning that the input variables, i.e., the set of independent and control factors taken into account, cannot determine well the value of the dependent variable. From a practical

perspective, this means that there might exist additional metrics that cover peculiar aspects of mobile applications and that can contribute to the explanation of the statistical power of the model—this seems to suggest the need for more extensive and comprehensive studies on the factors making mobile applications more defect-prone.

Looking at the table, a first aspect to discuss is the high significance of PRE_RELEASE_CHANGES (ρ-value <0.001). The estimate is positive, meaning that the higher the number of changes done before releasing the higher the likelihood that classes will be subject to defects. The result is not really surprising since the past history of a class has been found to strongly influence its future quality in a number of previous work [106, 142]. In this sense, our findings corroborate what discovered in the available literature.

Analyzing the effects of the other confounding factors, we found that (i) the presence of code smells and (ii) the size of production code can influence the number of POST_RELEASE_DEFECTS. As for the former, our results still confirm that the presence of design issues in production code, and of GOD CLASS and COMPLEX CLASS instances in particular, leads the affected classes to be more defect-prone [137, 226]. As for the latter, it is worth noticing that the estimate is slightly negative (-0.06): this indicates that the lower the number of production code lines, the higher the number of post release defects. While at first glance this could sound counter-intuitive, there are two observations to do. In this first place, the estimate is close to zero and, therefore, there might not be evident reasons making the metric connected to the dependent variable. In the second place, however, the result could be explained by a larger adoption of third-party libraries [267] that has the effect of sensibly reducing the amount of code in the app but, at the same time, increasing the likelihood of developers introducing defects because of API misinterpretation or the usage of defect-prone APIs—as shown in literature [22].

Turning the attention to our core interest, namely the impact of tests on POST_RELEASE_DEFECTS in mobile applications, we found that cohesion of test cases is a very relevant aspect (ρ-value<0.001) that influences the dependent variable with a positive estimate, i.e., the higher the cohesion the higher

the number of defects. Also in this case, the result is counter-intuitive. The findings of **RQ**$_1$ have shown that mobile applications are characterized by a few number of tests with a limited size: having a high cohesion in these cases may indicate that the test exercises only few and strongly cohesive functionalities in production code, hence neglecting others. The low coverage observed in **RQ**$_3$ seems to confirm that the few tests available are not able to verify the production code in an appropriate manner.

The significance of test smells confirm previous results achieved in the context of traditional applications [280]. Interestingly, we noticed that the INDIRECT TESTING smell has an estimate of 1.29, hence directly affecting the number of post-release defects. This smell arises when a test exercises multiple classes of the production code, not being able to focus on a specific target class. As previously shown [280], the lack of focus is among the key test-related problems increasing the defect-proneness of production code.

Last but not least, the COMMENT RATIO of test cases was found to be statistically significant, with an estimate of -15.43. This means that the lower the amount of documentation in tests the higher the number of defects in production. Our results in **RQ**$_2$ revealed that test cases of the considered applications are indeed poorly documented: the statistical analysis suggests that such a lack of documentation represents a serious threat to the reliability of source code. This is in line with previous findings [245] showing that non-commented tests cause a decrease of program comprehensibility and, for this reason, developers might encounter difficulties in detecting defects in production code.

Summing Up: Post-release defects are mainly influenced by the number of changes performed on production code. A few test-related-factors negatively contribute to the phenomenon as well. The cohesion of test cases is one of the most significant factors influencing the number of post-release defects. Other aspects such as the comment ratio and presence of test smells are still important but with a lower significance.

11.6 $\mathbf{RQ}_5$ - ON THE DEVELOPER'S OPINIONS ON MOBILE APP TESTING

This section reports on methodology and results that address $\mathbf{RQ}_5$.

11.6.1 *Research Methodology*

The analyses done so far provided a quantitative view on the state of the practice in mobile application testing. While we could provide some additional insights through our manual investigation of the considered apps, these are clearly not generalizable and would require further investigation. Being aware of that, our last research question sought to elicit the opinions of developers having a solid experience in the context of mobile app testing.

Previous work in the field have exploited survey research to complement mining studies (e.g., [212, 233]): by nature, surveys provide quantifiable data that can be used to establish, on a large scale, the maturity of a technology or, in the software engineering field, of novel instruments. Yet, surveys are not interactive and the data coming from them are likely to lack details or depth on the topic being investigated [266]. As the goal of $\mathbf{RQ}_5$ was to gather insights and in-depth opinions on the findings achieved in the mining study, we then preferred not to go for a survey, favoring a more qualitative approach like the one of focus groups, which are rarely large enough to draw definitive conclusions, but have the advantage of fostering discussions and uncovering ideas that otherwise would have been missed [266, 325].

More specifically, focus group research is defined as a small group of carefully selected participants who contribute to open discussions for research [325]. In the context of our study, a focus group enables a joint discussion on current testing practices and limitations among the recruited experts. From a methodological standpoint, the participants were invited to join an online Zoom meeting[6]—note that at the time of the study we were not allowed to run a physical meeting because of the COVID19 pandemic.

Two of the authors acted as moderators. At the beginning of the meeting, a 2-minute presentation of the participants was allowed, so that all of them

6 https://www.zoom.us/en/

got to know the others. Then, we provided an overview of the main goals of the study, a brief explanation of the research methods employed, and a detailed discussion of the results achieved in the context of $\mathbf{RQ}_1$-$\mathbf{RQ}_4$. To this purpose, a 10-minute slideshow was prepared: the last slide was designed to contain a summary of the main findings of the study and was kept shared with the participants till the end to allow them to always bear in mind the achieved results. It is worth noting that, while summarizing the results, we highlighted that these came out exclusively from open-source applications—this was done with the aim of setting the participants' expectations on the open-source side.

In the second part of the focus group, participants were asked to comment on the results and report experiences with respect to the testing of mobile applications that might explain the quantitative results of the study. This part of the meeting lasted 45 minutes and was kept by the moderators highly interactive: they did not simply leave the word to each participant, but asked others to comment and reflect on the possible reasons behind what s/he was reporting. The entire discussion was recorded and stored for analysis.

Upon completion of the recording, Zoom provided as output both the video registration and a text file reporting the transcript of the meeting. We reviewed the transcript together in order to identify the main insights and comments left by the participants. We finally addressed $\mathbf{RQ}_5$ by reporting the most relevant insights from the focus group.

Table 11.10: Background of the focus group participants.

ID	Dev. Exp.	Mobile Exp.	Working context
P1	13	10	Software Corporation
P2	11	6	Airline company
P3	11	6	Mobile Software House
P4	10	6	Researcher & Spin-off CTO
P5	11	6	Testing researcher

11.6.2 *Analysis of the Results*

Table 11.10 reports the background of the five experts to the focus group. As shown, all of them have a similar development experience and, since at least five years actively work on the development of mobile applications. Three participants (P1, P2, and P3) have a full-time appointment in large software companies of different nature: P1 works within a well-known US software and technology corporation founded in 1975, P2 is employed in a Dutch airline company, while P3 works for a multinational corporation that develops mobile applications. The fourth participant (P4) is a Senior Research Associated in an Italian university, but also has a partial appointment as Chief Technology Officer of a technological spin-off operating in the context of big data analytics. Finally, P5 is a researcher in the field of software testing having, however, experience in the development of mobile applications, i.e., P5 was employed within a mobile software company before starting the academic career.

For the sake of readability, in the following we report the key insights coming from the focus group by discussing each research question independently.

Commenting $\mathbf{RQ}_1$. After the introductory part, the two moderators started the focus group by asking whether the presented results were in line with the participant's expectations of how mobile testing is applied in practice. All participants agreed on the fact that, based on their experience, mobile apps are poorly tested and that, unfortunately, our results for $\mathbf{RQ}_1$ provide a representative overview of what happens in practice. In this respect, P1 commented that *"the majority of mobile developers have poor testing skills and, indeed, they mostly perform manual testing by adding pre- and post-conditions in the production code"*. In other words, according to P1's opinion, developers tend not to develop test cases but verify the behavior of their apps by crafting specific statements within the production code to verify pre- and post-conditions while developing or evolving the code. It also turned out that these statements might be even opportunistically disabled, i.e., test code is activated for debugging purposes and then commented when the app is finally released. The other participants agreed with the P1's take, yet they also pointed

out that this is a typical behavior observed with small applications developed by a few number of developers having low or no software engineering skills. P4 also found another motivation for the low amount of tests: in some cases, s/he said, *"testing specific usage scenarios is challenging, since mobile apps can interact with various hardware components and sensors (like the Bluetooth)"*. Hence, the overall discussion not only highlighted education challenges, e.g., how to address the problem of testing mobile applications at an education level, but also that developers sometimes need specific test beds or mocking strategies to simulate the behavior of external hardware components.

When it comes to the classification of test cases reported in **RQ**$_1$ (see Table 11.2), the participants unanimously agreed that the higher percentage of unit tests discovered in our study is due to the higher simplicity of performing unit testing with respect to the other types. Furthermore, P1 pointed out that *"the lower amount of integration and system tests might be also explained with developers writing tests that cover very specific, domain- and context-dependent use cases of their applications"*. Reasoning around this statement, it seems that the small number of integration and system tests may not necessarily be a problem in practice, since developers might have a deeper knowledge of the use cases that users will more frequently apply. Perhaps more importantly, P2 and P5 highlighted that mobile developers have the opportunity to test apps as a whole by employing established behavioral testing framework like CUCUMBER[7] and others—which are actually widely used in practice [167]. Interestingly, however, P2 reported that: *"behavioral tests provide developers with the perception that everything is working properly, but there must still be uncaught bugs. Nevertheless, in most cases developers accept those bugs since the cost of writing integration and system tests would be excessive"*. These observations allowed us to provide two main conclusions. On the one hand, having a few integration and system tests might not necessarily be an issue as long as they are complemented with testing frameworks that can exercise the app in different manner. On the other hand, our findings further support the work done by the research

7 https://cucumber.io

community around automatic test case generation: as a matter of fact, writing tests remain a costly activity that should be appropriately supported with automated mechanisms, especially in a context where continuous releases are expected to be delivered.

Our participants also actively discussed the results reporting non-functional attributes to be poorly tested. P3 argued that *"while there exist some frameworks to assist developers while exercising, for instance, performance and energy constraints, software companies do not often care about these types of testing"*, i.e., companies focus on functional requirements, neglecting non-functional ones. Moreover, P3 explained that *"it is hard to create non-functional tests because the definition of oracles is challenging and developers do not often have expertise to deal with the complexity of these tests"*. The other participants also pointed out a lack of tools able to *measure* non-functional aspects. To draw a conclusion, the discussion raised two main challenges for researchers: the need for more research on oracles and the need for more techniques/tools able to properly assess non-functional attributes of mobile applications - especially when these depend on external events.

Commenting RQ_2. Moving the attention to the developer's take on the results achieved when considering the design quality of the manual tests object of our investigation, the participants were quite interested in commenting on the documentation of those tests. P2 pointed out that *"is it not really surprising that amount of comments is low. There is a growing trend in industry for which developers should not spend too much time in documenting test cases since they will not frequently change over time"*. P5 confirmed this line of thinking and added that *"in the company I worked, the governance used to employ a strict naming convention to enforce developers write test names that clearly define the goals of the test and its target; in this way, failing tests could be easily retrieved and diagnosed"*. These observations led us to first argue that the documentation strategies for test cases are drastically different from those of production code, i.e., the focus is on names that can quickly evoke the responsibilities of the tests rather that on code comments that are more costly to write and may possibly become outdated. At the same time, we still see room for better assisting developers by means of improved

automatic code comment generators as well as refactoring instruments that can update tests and their documentation when new changes to production code are applied.

When addressing the readability of test code, participants were instead reluctant to consider this as a relevant aspect for design quality. In this respect, P2 reported that *"the readability of tests is much lower than the one of production code, but this is quite normal given the different goal that testing has"*. In other words, the participant argued that the main objective of test cases is to find defects, independently from how good the test code is readable. Additionally, P4 claimed that *"the readability values are really low considering that tests are usually short pieces of code. Perhaps, the metric employed does not appropriately capture the readability of tests"*. This statement led to the formulation of a hypothesis: the currently available test metrics do not properly measure the desirable properties of test cases. We further investigated such a hypotheses in the remainder of the discussion.

The participants also had concerns when discussing the structural code metrics. P3 reported that *"in my experience, low cohesion and high coupling in tests indicate that there is something wrong in production code"*; the other participants agreed with this statement and confirmed that the status of test cases often simply reflects the quality of production code. Going deeper into the discussion of the specific metrics, P2 added that *"the coupling values are particularly worrisome, it is likely that tests only exercise a few methods of the production code"*, while P1 reported that *"the test cohesion must not necessarily be high, yet this may indicate that there exist poorly cohesive test suites exercising more production classes"*. Based on these observations, it seems clear that there is a strong relation between production and test quality and, therefore, our results can reflect the more general poor quality of mobile applications—hence corroborating the findings by Linares-Vásquez *et al.*[166].

To conclude the discussion on $\mathbf{RQ}_2$, the moderators explicitly asked participants whether the available test code metrics are actually suitable to provide developers with relevant information about the design quality of tests. In response, P2 explained that *"there are some good metrics, like LOC, while*

others are quite meaningless in practice, like the comment ratio". More in general, P1 reported that *"the level and goal of the metrics that we expect to assess tests is different from those of production code"*. Although our findings in this respect are not conclusive, we believe they raise the need for further investigations into the real usefulness of the current metrics.

Commenting RQ$_3$. The participants went through the results achieved when computing code coverage and assertion density. In the first place, all participants agreed that the results are not surprising: these are, indeed, in line with their expectations since *"it is a standard, yet unhealthy practice that of considering test cases as second-class citizens"*, said P5.

P2 added that *"most of the tests analyzed are at unit-level, which makes the low coverage even more worrisome: they are likely to exercise only the easiest parts of the production code"*. More in general, P3 commented that *"even if the coverage is objectively low, this metric does not necessarily imply that the tests cannot catch defects"*.On the one hand, this confirms recent findings by Grano *et al.*[102], i.e., test case effectiveness is a multifaceted concept that should be assessed by combining multiple metrics. On the other hand, the participants' opinions led us to further push the need for additional investigations into the definition of novel metrics to assess test code quality and effectiveness.

The discussion on assertion density led to a similar conclusion. P1 reported that *"some tests might be useful even when they have no assertions; the assertion density is a metric, but not necessarily good"*. In addition, P1 and P4 observed that the metric computation is naturally biased by the amount of code required to setup the test environment, i.e., if a test requires more code to prepare the environment the denominator will be higher, leading to a lower density that does not necessarily indicate the poor quality of the test.

Concluding the discussion for these results, we could confirm that the general feeling of our participants was that the available testing metrics are not enough to provide a comprehensive view of test code effectiveness.

Commenting RQ$_4$. When discussing the results on the relation between test code properties and post-release defects, participants basically confirmed the opinions given for the previous research questions. In the first place, they

pointed out that test cases of such a poor quality cannot provide any significant indication to developers and are, naturally, going to fail in catching defects in production code. They also clarified how the continuous changing nature of mobile apps make the testing development process particularly challenging, since tests must be frequently updated and, as a matter of fact, there is typically not enough workforce, time nor experience to write effective tests and deal with the intrinsic complexity given by the environmental constraints (e.g., hardware components and sensors).

Summing Up: The quantitative results of our study reflect the expectations that developers have of the status of mobile app testing. The participants provided further explanations and insights into the matter, e.g., by raising specific education and technical challenges that the research community should carefully look at. In addition, our focus group let emerge the need for additional/novel metrics able to better measure both quality and effectiveness of test cases.

11.7 CONCLUSION

In this chapter, we conducted a large-scale investigation into the characteristics of test suites written by developers of mobile applications under four perspectives, namely (1) whether and to what extent these apps are tested and which kind of tests are developed, (2) what is the design quality of the test suites, in terms of code metrics and test smells, (3) what is the effectiveness of tests, considering assertion density and code coverage, and (4) how test-related metrics are associated to the defect-proneness of production code. The quantitative insights coming from the analysis of these aspects were then discussed in the context of a focus group involving 5 mobile testing experts, who commented the achieved results and provided practical explanations and experience reports useful for understanding the status of testing in mobile as well as the key limitations that must be addressed.

The main results of the study highlight that 40% of the considered apps have at least one test suite; developers mostly test source code to exercise its functionalities, while other types of testing are less widespread. Test smells

represent a key problem for most of the test suites, since some of them exhibit characteristics making them possibly flaky. Their effectiveness is low when considering all the computed metrics. Finally, the characteristics of test cases lead to a negative impact on production code and, indeed, most of the statistically significant test-related factors in our study are correlated to a higher defect-proneness of the corresponding classes. These findings reflected the expectations that the involved experts had when thinking of the status of mobile testing. Furthermore, from the focus group discussion we could delineate a number of education and technical challenges that future research should address.

12

THREATS TO VALIDITY, DISCUSSION, AND IMPLICATIONS

This chapter reports some aspects that might have threaten the validity of the results achieved in our empirical studies and discusses our main findings to answer $\mathbf{RQ}_c$ and $\mathbf{RQ}_d$.

12.1 THREATS TO VALIDITY

The results of our studies might have been biased by a number of factors. In this section, we overview the main threats to validity and how we mitigated them.

12.1.1 *Threats to Construct Validity*

Threats in this category are concerned with the relation between theory and observation. A key point in this regard is related to the accuracy of the tools used to compute the metrics for the two studies. This threat has been mitigated by the selection of tools that are (i) well established in the field (e.g., DECOR [197] for the detection of code smells) and (ii) accurate enough for conducting our study, according to the manual validations conducted in the context of our work. However, it is worth mentioning that the manual analyses could only deal with false positives but not with false negatives.

A partially different discussion should be made when considering test smells. To detect them, we relied on the detector made available by Bavota *et al.*[25]. While previous studies have shown that its F-Measure is close to 86% [25, 239, 241], the detector could have had a different accuracy in our context. Recognizing this as a possible threat to validity, we conducted an additional

investigation aimed at measuring the precision of the detector.[1] Unlike the case of code smells, we could not validate all the 1,217 instances output by the test smell detector as a manual analysis would have been excessively expensive. Instead, we focused on a stratified statistically significant sample (confidence level=95%, confidence interval=5%) composed of 169 instances. The task was jointly conducted by two inspectors and consisted of assessing whether each test smell candidate presented a certain design issue. At the end of the process, 86% of the instances were considered as real test smells - thus confirming the high precision of the detector.

We are also aware of the possible limitations of the SZZ algorithm that we adopted to identify commits where defects were introduced, as highlighted in recent works [265]: in this respect, we conducted a manual validation of the results of the SZZ algorithm that aimed at excluding false positives. Of course, we are aware that this analysis could not cope with false negatives.

Another discussion point relates to the methodology employed to link production classes to test cases: in particular, we employed a traceability technique based on naming conventions, i.e., it identifies the test corresponding to a certain production class by looking at the name of the test and verifying whether it is the same as the production class expect with the prefix '`Test`'. While the accuracy of the technique has been previously assessed [310] showing a good compromise between accuracy and scalability, the linking procedure may have introduced some bias in cases tests exercising a production class are not all included in the test suite retrieved by the technique but put in other test suites. In our case, this may have been happened, as mentioned by the interviewed developer of Apache Commons-Pool who commented on our findings in the context of our additional qualitative analysis.

There are two observations to make with respect to this potential bias. First, it has been pointed out by only one developer and was related to only one of the considered projects: as such, we are not able to estimate the extent of this bias in other systems as well as to verify whether this may have represented a general problem for Commons-Pool or if it was instead focused on a subset of classes of the project—it is worth noting that a manual examination of this

1 The recall cannot be assessed because of the lack of an oracle.

bias would have not only been prohibitively expensive, but also error-prone given our lack of expertise on the project. Second, we could not identify an alternative traceability technique which may have provided better results than the one employed: indeed, while some more sophisticated test-to-code traceability techniques have been proposed [247], these are likely to suffer from similar issues as the one based on naming convention. As an example, the slicing-based approach proposed by Qusef *et al.*[259] exploits slicing and conceptual coupling to identify the set of test suites associated with a production class. By design, this approach may have higher recall, since it is able to model the case in which more test suites exist for a production class. At the same time, however, this may not be enough. The involved developer mentioned a finer-grained problem where specific test cases are included in other suites, as opposed to the existence of multiple test suites for a production class. As such, the technique by Qusef *et al.*[259] may lead to overestimate the number of tests for a certain class, decreasing the precision of the analysis. In other words, such a fine-grained linking between test suites and production classes would have needed a traceability approach able to cluster the test cases connected to a production class: unfortunately, to the best of our knowledge, such a technique is not available in literature—we hope that this additional finding may serve as an input for the software traceability research community.

Furthermore, to extract a comprehensive list of test-related factors to experiment, in the fist study we applied a MLR [94]. In this regard, possible threats refer to the soundness and completeness of the review. With respect to the former, two inspectors followed well-established guidelines [143, 326] to search, analyze, and select relevant sources; moreover, the joint work conducted by the two authors have reduced the risk of subjective evaluations of the resources to include as well as allowed a quick solving of possible disagreements. As for the latter, we defined a search query targeting the research goals of the study; at the same time, we targeted databases that allow searching for most of the white papers published in our community. Furthermore, we analyzed all the relevant GOOGLE pages when gathering gray literature, also performing it using the incognito mode to avoid biases

due to previous navigation history. However, it is worth noting that, being very specific, our search query could have led to overlooking some potential sources; indeed, the resulting number of retrieved sources is quite limited. Anyway, we think that this just represents a marginal threat to the study validity since a wide set of heterogeneous factors was retrieved by the MLR process.

Finally, previous work has found that some of the applications available in the F-Droid repository are very basic projects [97, 98], thus possibly biasing the conclusions of our second study. To overcome this limitation, we manually went over each of the initially downloaded apps in order to discard those that appeared to be too trivial to be considered. In particular, we looked at their repository in order to check whether they result active, e.g., in terms of commits, conjecturing that trivial apps are not updated and actively developed, e.g., since could be part of a university project for an exam.

12.1.2 *Threats to External Validity*

With respect to the generalizability of our findings, in our first study we analyzed eight open-source Java projects. Analyzing only a small number of systems could threat the external validity of our study. However, during the context selection, we had to deal with a set of constraints that have significantly limited the list of candidate systems. In particular, we restricted our search to Maven projects with no sub-modules and a standard Maven structure. We made this choice to allow the employment of the PiTest command-line tool for the calculation of the dynamic factors (i.e., line and mutation coverage). Nevertheless, further replications of our study, conducted in different contexts overcoming the constraints mentioned above (e.g., by aggregating the results of multiple submodules) might be worthwhile to corroborate our findings. In the second study, instead, we targeted a large set of open-source applications, thus allowing the verification of the characteristics of tests on a large scale. Nevertheless, it is worth pointing out that our findings may differ in different contexts, e.g., in closed-source apps testing practice results different, as well as settings, e.g., when considering test smells other than those taken into

account. As such, further replications of our study would be desirable and are already part of our future research agenda.

Moreover, it is worth remarking that the statistical models were built over the specific independent, control, and dependent variables adopted in the two studies. Despite our effort in taking into account all factors known to have an impact on post-release defects, we are aware that different results may arise when considering different/additional variables. Similarly, our findings on the relation between test smells and software quality are deemed to be valid for the five test smell types considered in the study: other results may be achieved with other design issues.

Another aspect to consider when interpreting our results is that some post-release defects may be discovered with unit testing, while others can be found only at higher-levels (e.g., with integration or system testing) [66]. We are aware of this point and recognize that our study is limited to the analysis of the behavior and the relation that unit tests have with post-release defects. Replications of our studies targeting different test levels would provide a more comprehensive view of how tests can forecast post-release defects. On a similar note, our study investigated the role of the tests actually available in the considered software projects: it may be possible that different results could be achieved in cases where the diversity of test cases, i.e., the extent to which a test is different from the others in the suite [76, 77], is higher. Understanding the impact of diversity on both test and production code quality is part of our future investigations.

12.1.3 *Threats to Conclusion Validity*

As for the relationship between treatment and outcome, a first possible threat is connected to the statistical models built in our studies. Throughout our research, we controlled the impact of independent variables for possible confounding effects due to the characteristics of production code, considering both product and process metrics that have been shown to be connected with the dependent variable.

Another threat is related to the actual suitability of the employed statistical method, i.e., Generalized Linear Model. In this regard, before selecting it we verified the assumptions that the model makes on the underlying data. Nevertheless, it may still be possible that the statistically significant variables discovered through the use of linear regression may be due to the specific data manipulation and analysis done by the statistical model [120].

Another potential threat could be related to the manual analysis we performed to classify granularity and type of tests in mobile applications. To this aim, we followed a grounded-theory approach [255] where two authors first classified an identical set of tests in order to tune their judgment and proceeded with the classification process smoothly. Of course, we still cannot exclude the presence of some imprecision in the classification, however the high agreement reached by the inspectors makes us confident of the reliability of the process conducted.

As a final remark, in our second study we follow a focus group methodology. This is a qualitative research method that, by nature, does not require the participation of a large amount of experts — it is explicitly designed to have a small group of participants able to foster discussion and provide insights/recommendations on the phenomenon of interest [266, 325]. The conclusions reached might not be definitive: we are aware of that, yet we preferred to have in-depth opinions on the findings achieved in the mining study rather than more generic results that might have achieved using other instruments, e.g., surveys. As usual, however, replications are desirable and might provide complementary insights into the testing of mobile applications.

12.1.4 *Threats to Internal validity*

This category of threats concern with intrinsic factors of our studies that could have influenced the reported results. In this regard, there are some intrinsic issues when computing some of the test-related factors, like mutation coverage. In particular, there are two relevant aspects when measuring this metric: the problem of equivalent mutants and the one of live mutants. As for the former, it arises when two generated mutants are semantically equivalent.

Unfortunately, determining whether a mutant is equivalent is an undecidable problem. Furthermore, detecting them is also hard - there is still no mature tool available for this task [175] - and computationally expensive [218]. For these reasons, equivalent mutants represent a common threat to the validity of the results of studies concerned with mutation testing. In our study setup we took into account the equivalent mutants problem when selecting the mutation testing tool. Among the available ones, PITEST has shown better performance than others: specifically, less than 20% of the mutants generated by the tool are deemed to be equivalent, which represents an important step forward in the context of mutation testing [79]. Other control mechanisms, e.g., manual removal of equivalent mutants, would not be feasible in our case because of the high number of mutants generated by PITEST.

As for the live mutants, these represent the mutants generated by the tool but not detected by the available tests. Live mutants allow the mutation score computation (i.e., mutants detected over mutants generated). On average, these mutants represent around 30% of all mutants generated, as shown in Section 10.2.3, meaning that most of the tests in our dataset have a rather high mutation score. This suggests that the study takes into account valuable tests that can actually be used to investigate post-release defects.

12.2 DISCUSSION AND IMPLICATIONS

This section discusses the main findings of the studies presented in Part II in response to RQ_c and RQ_d.

12.2.1 *RQ_c - On the relation between test-related factors and software code quality*

The results of the studies in Chapter 9 and 10 provided two main findings to be further discuss.

On the (limited) importance of test-related factors for software code quality. The main outcome of our research reports that—surprisingly—most of the considered test-related factors do not have a significant explanatory

power with respect to post-release defects. Despite this could seem strange, it is possible to reason on why this could happen. Let consider a scenario in which tests have good quality and effectiveness: these tests are likely to accurately identify defects present in the same snapshot of the production code, however they do not necessarily predict the future defect-proneness of the class under test. So, probably the test-related factors commonly known and used in literature are not appropriate enough to catch the defect proneness of the exercised code. Proposing new metrics able to describe the relation between tests and software quality could be an interesting cue for future works.

From another point of view, there are some exceptions that may allow us to claim that keeping test code design under control might still help developers in reducing the number of post-release defects. In the first place, the test LOC metric not only appears as highly significant, but it is also inversely proportional to the dependent variable: this indicates that larger tests (that are likely to exercise deeper the production code) reduce the risk of having defects in future versions of the system. Thus, our findings seem to indicate that the lines of code of a test suite may represent a proxy measure for test-code effectiveness.

Furthermore, while other test-related factors investigated in the study (e.g., line coverage or assertion density) are not correlated enough to test LOC to cause collinearity, it is reasonable to believe that they have some sort of relations with the size of the test: for instance, the assertion density generally tends to increase with the size of the test [154]. On the one hand, these relations should be further assessed in the future. On the other hand, this observation may further suggest that high-quality tests lead to post-release defect reduction: the results achieved on the role of test smells, particularly *Mystery Guest*, also go toward this conclusion.

In order to better comment on our results, we performed an additional qualitative analysis in which we contacted the top contributors of the considered projects. In so doing, we followed a similar experimental design as Mäntylä *et al.*[180]. In particular, we sent direct e-mails to the two top developers of the systems, i.e., the two having the highest amount of commits, asking them

to comment on our findings and provide feedback on some boundary cases we discovered when analyzing their systems—this means that developers were inquired only on the matters related to their own projects. Unfortunately, we received an answer for just one of the considered systems—even tough they were insightful to better contextualize and understand the findings of the study. The answers was related to COMMONS-POOL. In this specific case, we asked the developer to comment on the release 2.3 of the project, in which the class named `BaseGenericObjectPool` had 849 lines of code, while the corresponding test suite `BaseGenericObjectPoolTest` had just 43 lines of code. After release 2.3, the class `BaseGenericObjectPool` had 23 defects. At a first sight, this may suggest that the test suite was not robust enough in preventing or diagnosing the introduction of defects. However, the developer found that just considering the test suite `BaseGenericObjectPoolTest` could potentially be not enough. He pointed out to us that when code is refactored, the tests are left in the original test suites to help detect regressions during the refactoring. So, there could exist a subset of tests in other classes, that we did not consider, which exercise the production class `BaseGenericObjectPool`—the test-to-code traceability technique exploited in the study may have under-estimated the number of tests connected to the production class.

To sum up, our findings reveal that the problem of understanding the effect of tests on post-release defects is still open and would require further investigations—especially in the lights of the potential threat to validity related to the test-to-code traceability technique raised by the involved developer. At the same time, the results provide some hints of the importance of (i) test-related factors for software quality, even though other aspects, e.g., the number of changes to production code, are still primarily connected to post-release defects and (ii) continuously keep test suites up to date with the changes applied to production code.

On the comparison with previous studies. A number of researchers have investigated the role of test-related factors on post-release defects in the past, finding them as highly relevant. While our results do not tell the opposite,

we found that most of the considered factors have a lower explanatory power than the one previously reported.

The key to explain the difference between our outcomes and the ones previously provided is in the presence of confounding factors that possibly balanced the effect of test-related factors. This is particularly true in the case of LOCs of production class and pre-release changes, that are the most relevant metrics to explain the future defect-proneness of source code and are directly proportional to the dependent variable, i.e., the higher the size and the number of pre-release changes, the higher the number of post-release defects. This suggests a pretty straightforward interpretation: classes having large size or being involved in several changes over the history are more prone to have defects in the future.

Our results provide a number of implications for both research community and practitioners.

Keep the change process under control. The most important finding of our study is the very high influence of pre-release changes on post-release defects. This relation indicates that the change frequency of classes impacts the future defect-proneness of production code more than other aspects, confirming previous findings on the relationship between change- and defect-proneness [52, 119]. In this respect, it may be possible that high-quality pre-release changes prevent the emergence of post-release defects: we indeed noticed that the LOC of production code—which has been often used as a proxy metric for code quality—and pre-release defects are statistically significant factors for the future defect-proneness of source code. Based on these observations, we can claim that keeping the change process under control would be worthwhile and that the definition of mechanisms supporting developers when dealing with software evolution represents a key challenge for the research community. While a number of attempts in this direction have been performed during the last years, e.g., through the definition of of just-in-time quality assurance mechanisms [130, 246, 248], we believe that further research effort should be invested. In particular, most of the approaches developed so far should be considered as prototypes and, as such, are still not mature enough to be used in practice. For example, researchers have been

working on just-in-time defect prediction models (e.g., [130, 248]), but up to now there are no fully-available tools that enable their practical usage. This clearly represents a key threat to the adoption of these tools in practice that the research community should investigate more. As a consequence, we argue that continuous integration pipelines as well as typical software development practices should be empowered with additional instruments that allow developers to promptly assess the quality of the changes made on production code: for instance, we refer to defect localization tools that can be integrated within CI environments or code smell detectors and refactoring recommenders that allow an agile quality improvement of source code during the code review process.

At the same time, tool vendors have been spending effort in providing developers with tools that can help them spotting defects. The main outcome is represented by automated static analysis tools, which are generally used in open-source systems as highlighted by a number of papers in literature [315, 320, 336]. One of these tools is FINDBUGS, which is the one employed by the APACHE SOFTWARE FOUNDATION and, as a consequence, by the projects considered in our study (this tool is configured within the APACHE CONTINUUM server they have in place). On the one hand, static analysis tools suffer from a high rate of false positive alerts [127]: this aspect has the effect of reducing the trust of developers with respect to the outcome of these tools, possibly leading them to ignore relevant defect warnings. On the other hand, it has been shown that open-source systems (including those of the APACHE COMMONS family) do not apply continuous code quality practices [314], meaning that they do not run quality checks at every build they do: this is an additional limitation of the current quality assurance practices. According to these observations, our work further stimulates the research effort around the definition of techniques able to reduce the number of false positives given by static analysis tools as well as mechanisms enabling the adoption of continuous code quality.

Test-related factors and defect prediction. The results of the study revealed that, in some cases, test-related factors are related to post-release defects. While we cannot speculate on whether there exist specific types of defects

that can be better analyzed through the exploitation of test-related factors, we still see some value in this finding. More specifically, from our study we observed that test-related factors become relevant especially when process metrics are not considered. This represents an interesting case for defect prediction: indeed, new projects interested in deploying these models might not have enough historical data to enable the computation of process metrics. In these cases, there are two solutions. On the one hand, developers may rely on cross-project information to train defect prediction models [340]: nevertheless, the adoption of this strategy does not still provide accurate results, hence limiting its applicability. On the other hand, developers can create prediction models based on product information coming from the analysis of their own systems: our results can be useful in this context, as test-related factors may complement other product metrics and potentially improve the quality of the predictions. In the recent past, researchers have started looking at the role of tests in defect prediction [31], however we believe that our study may inspire further research on the matter, especially based on the factors that turned to be important for post-release defects, e.g., test size or presence of five test smells considered in the study.

Test-related factors and automatic test case generation. Our findings point out that other test-related factors, namely test size and test smells, are more related to post-release defects than metrics generally considered relevant, i.e., code or mutation coverage. This seems to suggest that the design of test suites matter. This may possibly pave the way for the next generation of automatic test case techniques that do not consider anymore (or decrease the importance of) code and mutation coverage as main metrics to optimize during the creation process: such a generation mechanism would possibly allow automatic tools to focus on the creation of tests around factors that are more connected to post-release defects (as also proved by Kochhar *et al.*[145]). Furthermore, our results also highlight the existence of production-related factors, and specifically production code size and presence of code smells, that have a relation with post-release defects. It would be worth to consider how these production-related factors can contribute to the generation of effective test classes: for example, existing automated tools may exploit these

metrics within their fitness function and balance them with other metrics with the aim of refining or further optimizing the generated test suites around the metrics that are more connected to post-release defects.

12.2.2 RQ_d - *Testing activities in mobile applications*

We addressed RQ_d in Chapter 11. The achieved results provided a number of insights and practical implications for the research community that need further discussion.

Mobile apps contain very few numbers of Java tests. The first evident, worrisome result of our study clearly indicated the lack of tests in mobile applications: not only the mean number of tests is ≈ 3, but also the percentage of apps without any test is rather high (60%). There are multiple factors possibly contributing to this finding. First, our dataset is composed of open-source mobile applications that can be developed under different conditions with respect to other applications: as an example, they can be developed by inexperienced or novice programmers with little knowledge on testing practices [319]. During the focus group discussion, our participants also raised this point and commented on how the lack of software engineering/testing expertise can have a significant impact on how mobile applications are tested. At the same time, the developers involved reported that test cases are typically seen as second-class citizens, especially in a dynamic environment like the one of mobile development, where a continuous release model might soon make tests outdated other than increasing the cost required to maintain and evolve them. In this respect, our findings corroborate previous results obtained by Beller *et al.*[27] on the lack of developer's willingness in successfully evolving tests. As an exemplary case appearing in our dataset, let consider the case of ACASTUS PHOTON,[2] an online address/POI search for navigation apps. Looking deeper at its issue tracker and the developer's comments, we noticed that the developers of the app have consciously postponed some testing activities with the aim of entering the market faster or because of the lack of time to dedicate to testing. For instance, in one of the issues still open

2 https://f-droid.org/en/packages/name.gdr.acastus_photon/

on the issue tracker (#2), one of the core developers of the app posted the following comment:

> *"[...] I'm probably going to merge the build changes later on too. [...] I don't have time to test them right now so just merging master."*

As shown, in this case the developer decided not to test the newly committed code change because of the need to other modifications to the production code. Even without an extensive search, we found similar cases in other apps of our dataset. Finally, our findings can be also due to the limited automated support that developers have when testing their apps. As pointed out in the context of our focus group, mobile developers experience very specific challenges when developing test cases, like the need for considering external events coming from hardware components or sensors. By looking at the state of the art, there exist a number of tools to automate GUI testing (e.g., MONKEY or SAPIENZ [182]), other than some frameworks for behavioral-driven testing, yet only a few automated and practical mechanisms are available for the generation of functional and non-functional test cases (e.g., EVOSUITE [89]). In addition, these tools do not explicitly take into account the problem of mocking hardware components. As such, our findings do not only support the research in the field of automatic test case generation, but also call for the definition of mobile-specific instruments.

On integration and system testing. According to our findings, most of the test suites present in mobile apps pertain to unit testing, while we discovered only a limited amount of them referring to integration and system testing [17]. This result might be due to various reasons. While the lack of automated tools and/or support mechanisms might influence the way mobile developers verify their apps for integration- and system-level faults, it is also worth remarking that different verification mechanisms, like manual validation, crowd-testing [158], or even the use of external tools that can exercise the apps to verify certain specific properties (e.g., energy consumption [70, 123]), might be put in place. As an example, our study highlighted that, depending on the context, the lack of automated integration and system JAVA tests might not necessarily

be a problem because developers may want to test the use cases that users perform more often when using the app, hence leading to the application of non-systematic or opportunistic testing strategies that are not automated, e.g., crowd-testing [158]. On the one hand, our findings may possibly pave the way for novel smart techniques that can analyze runtime usage logs to recommend when to add test cases or suggest modifications to the current ones. On the other hand, we point out the need for additional investigations into the methodologies employed by developers to perform integration and system testing.

Enabling testing of non-functional attributes. Most of the tests developed in mobile apps relate to functional aspects of production code, while few of them refer to testing of non-functional attributes like, for instance, energy consumption, security, or performance. The developers involved in our focus group clearly pointed out how hard the development of these tests can be. There are two key limitations of the state of the art in this respect. In the first place, defining an oracle for these types of test is a challenging or even a non-deterministic task, e.g., the oracle of an energy test must necessarily take into account the non-determinism of energy measurements. In the second place, our study highlighted a worrisome lack of instruments that support developers when measuring non-functional aspects: these have been often connected to the commercial success of mobile applications [22, 233, 237], making the lack of testing a threat for the overall sustainability of these apps. On the basis of these results, we therefore argue that more methods to manage the complexity of non-functional attribute testing should be designed: while the research community has been working already on the measurement side (e.g., [70]), to the best of our knowledge there is no study targeting the oracle problem when considering non-functional testing.

The design quality of mobile apps is low. Our findings report that most of the tests analyzed are affected by some form of test smells. Previous researches have shown how these problems can turn into critical threats to the effectiveness of tests [280]. To identify them, some test smell detectors have been developed in the past [107, 241, 311] and experimented in the context of mobile applications [250, 251]. Yet, there is no empirically-assessed

technique available to automatically refactor test code. As such, the practical support provided to mobile developers is still very limited.

On the need of novel metrics for test code quality. When analyzing the quality of test suites, we also computed code metrics capturing cohesion, coupling, complexity, and documentation aspects. Our quantitative analyses revealed a contradiction between the metric profile of tests and the actual presence of design issues. While the values of the metrics would not indicate problems with the design of test cases, we discovered that test smells are often present and lower the maintainability and understandability of tests. By contrasting these results with those achieved in our qualitative investigation, we discovered that the meaningfulness of these metrics is limited. The involved developers not only presented practical scenarios where the metrics could not be relevant (e.g., companies may implement guidelines for naming conventions, discouraging developers to write code comments), but also pointed out that the level and scope of test metrics are different from those of production code. Our findings indicate that researchers should go beyond the currently available code metrics and define novel indicators that can better quantify the quality of test cases. Furthermore, on the basis of our results we argue that both available and prospective quality metrics should be re-contextualized for mobile applications, for instance by providing an easy way to quantity how much the quality of tests depend on the quality of production code.

On the effectiveness of test suites. Another relevant finding is the low effectiveness of the test cases analyzed when considering both code coverage and assertion density. On the one hand, these metrics have been previously positively correlated to the fault detection capabilities of tests: as such, their low value is somewhat worrisome and depict a critical situation for the mobile applications considered. On the other hand, however, developers raised some practical critiques to those metrics: the assertion density computation might be biased by the amount of code required to setup the test environment, while the coverage only provides a part of the whole story. These observations further confirm the need to establish novel methods to evaluate test cases. Moreover, the results corroborate recent findings showing that the effectiveness of tests

represent a phenomenon that goes way beyond the currently available methods [102]: as such, we argue that newly proposed metrics should consider the aspects deemed important for developers and, more importantly, possibly be directly assessed against the developer's perception of test code effectiveness.

Teaching software testing. To some extent, both the quantitative and qualitative findings of our study showed that mobile developers do not have proper testing skills. This is even worse when considering the constraints that mobile apps might have, e.g., the interaction with sensors. As a consequence, our study highlights the need for interventions on an educational level. This concerns both software engineering courses covering basic testing skills, and more sectoral courses on mobile development providing students with specific mobile-oriented skills: we not only argue that the focus on testing practices should increase from a technical perspective, but educators should further pushed on the value of having quality and effective test suites in practice, for instance by showing the extent to which testing is connected to the failure of software engineering projects [291] or by designing additional seminars on the matter, trying to engage students with practitioners.

Part III

CONCLUSION AND FURTHER RESEARCH DIRECTIONS

13

CONCLUSION

The software engineering research community has been dedicating more and more attention to technical debt, its consequences, and the cause that lead to its introduction. Despite the noticeable effort spent so far, there are some aspects that are still unexplored, as well as some limitations that are not been overcame yet. In this thesis, we put our focus on two specific technical debt-related topics, namely code smell detection and technical debt in test code.

We firstly deal with the detection of code smells, which represent one of the most common forms of technical debt. Our aim was to mitigate the limitation of existing code smell detection heuristic techniques, i.e., (i) subjective developers' interpretation of their outcomes, (ii) low agreement between different detectors, and (iii) performance strictly related to thresholds definition. In this regard, our solution was to investigate the application of machine learning-based techniques for code smell detection. This led us to our first study [C01] (Chapter 3), in which we compared the performance of heuristic and machine learning-based techniques on a manually-validated dataset composed of 125 releases of 13 open source projects, considering five code smell types in a within-project scenario. Results of these preliminary studies highlighted that machine learning-based approaches are still immature to be applied in practice.

More importantly, our preliminary results also highlighted three main limitations of ML-based code smell detection techniques:

1. ML-based techniques suffer from the highly unbalanced nature of code smell datasets;

2. The set of metrics exploited so far could be too limited to allow automatic identification of code smells;

3. Similarly to heuristic techniques, also with machine learning, there is still a developers' subjective interpretation of the results provided.

This thesis faces these three limitations separately. In particular, to address the first limitation, we conducted a large empirical assessment on the role of data balancing for code smell detection [C02, J01] (see Chapter 4). In this study, we experimented with five different data balancing techniques and compared their performance also with a baseline ML classifier trained without the application of any data balancing technique. Results reported that SMOTE is the data balancing technique that leads to the best, but still limited, performance improvement. However, considering the limited improvement together with the overhead introduced by data balancing, in some cases could be preferable to not apply data balancing at all.

With respect to the set of metrics to use as predictors, we considered the introduction of other, not yet explored, metrics to predict the presence of code smell instances in the source code component. In Chapter 5 we presented a machine learning-based code smell detection technique that uses the warnings generated by three static analysis tools (i.e., Checkstyle, PMD and Findbugs) as predictors [C04, J06]. The results that emerged from a first preliminary analysis were really promising, indicating a drastic improvement in detection capabilities. However, more in-depth analyses conducted on a larger dataset have shown strong limitations also for this technique, as for the previous ones. Despite the poor limitation capabilities of the models built on the warnings generated by a single static analysis tool, the results show a significant improvement when these warnings are combined with each other or with other metrics used in the past.

Finally, to tackle the problem of subjectivity, we adopted an alternative strategy, presenting a technique able to predict the presence of code smell as actually perceived by developers [C03] (Chapter 6). In this study we used a set of 20 factors, related to different aspects of software development (ie, Product metrics, process metrics, developer metrics, and code smell metrics), to predict the criticality of code smells as perceived by developers. To measure this aspect, over a period of 6 months we sent a total of 1,733 e-mails to 372 different developers asking them if they perceived the presence of code

smells within newly modified classes and if so, to evaluate their criticality. on a scale of 1 (very low) to 5 (very high). The prediction, calculated through the use of different machine learning techniques, showed results, on average, 20% more accurate than the considered baseline.

After having conducted a comprehensive analysis of the problem of code smell detection using machine learning, we moved our attention to technical debt in test code. Specifically, in Chapter 10, we first conducted a multivocal literature review (MLR) aimed at collecting all the test-related factors associated with software code quality in the past [C06, J03]. Then, we built a statistical model to find significant relations between those factors and software code quality, in terms of post-release defects. Results of this statistical analysis evidenced that test-related factors generally have a way lower impact on software code with respect to other process-related aspects. Furthermore, a more important outcome provided by this study is that some intrinsic characteristics of tests, such as size and presence of test smells, have a higher impact on production code quality with respect to other coverage metrics considered important in the past. Other than analyzing standard systems, we also wondered about the attention that developers pay to test in mobile applications [C07, J04]. Chapter 11 reports a large empirical assessment in which analyzed a dataset composed of 1,780 Android applications to assess (i) the quantity and type of test cases present in Android applications, (ii) the quality of the test cases, and (iii) the effectiveness of the tests. From the results obtained, it appears that over 60% of the applications analyzed do not contain any test cases, highlighting little attention to testing activities by mobile application developers. As for applications containing at least one test suite, results show that developers pay more attention to functionality testing rather than other aspects (e.g., performance, GUI). Furthermore, it has been shown that the analyzed test cases have low effectiveness, with code coverage percentages around 20%.

13.1 LESSON LEARNT

The main lessons learnt from this thesis can be summarized as follows:

1. **Lesson 1.** *Heuristic techniques for code smell detection are too limited to be applied in practice.* One of the main and most surprising results of the study in Chapter 3 relates to the low performance of existing heuristic code smell detectors. Specifically, while they are able to achieve a very good recall, i.e., a high number of actually smelly instances are detected as smell, we found the precision to be extremely low, i.e., they generate a high number of false positive instances. These detectors are all built on the definition of detection rules in which a set of metrics is compared with strict thresholds. However, defining either the metrics to use and the thresholds before is a very subjective activity and, as also results demonstrate, it appears not to be generalizable enough.

2. **Lesson 2.** *Machine learning-based techniques are still immature for code smell detection.* This thesis provides a very deep analysis of the adoption of machine learning for code smell detection. We conducted a preliminary study in which we compared machine learning with heuristic techniques and then we individually treated all the major limitations identified. Despite the great effort spent so far, none of the proposed solutions seems to work properly to be applied in practice. A possible explanation of the low performance could be related to the subjectivity of the problem itself. Indeed, even humans could not be able to objectively define a ground truth of actual smells to be used as training set for Machine Learning-based algorithms. As such, even the manually validated data sets of code smell instances could suffer from a subjective interpretation by the human rater.

   All these consideration let us think that it might make sense to completely change the route and consider the adoption of new techniques for detecting code smell instances in source code (e.g., deep learning).

3. **Lesson 3.** *Developers pay low attention to testing activities.* Regardless of the context, open-source developers appear to pay way lower attention to testing activities with respect to other development activities. The repositories we analyzed are generally characterized by few tests,

having low quality and low effectiveness both in the context of standard and mobile applications.

4. **Lesson 4.** *Test code quality matters.* Despite the low attention developers spend in testing activities, our results evidenced that developing good-quality test suites can provide important benefits for preventing future faults in software systems. Indeed, in some cases, test code quality appears to be even more impactful than other metrics considered important in the past for preventing bugs. This finding suggests researchers give more prominence to test code quality, by taking into account this aspect also in other areas such as defect prediction or automatic test case generation. Moreover, practitioners should be pushed to pay more attention to test suites development.

13.2 OPEN ISSUES

Despite the effort devoted by the research community and despite the advances proposed in this thesis, the current state of the art still propose a number of open issues and challenges that need to be addressed in the future.

1. **Open Issue 1.** *Machine learning for code smell detection.* This thesis demonstrates that machine learning can overcome the limitations of heuristic techniques for code smell detection. However, at the same time, machine learning comes with its own limitations that do not allow its application in practice. Although this thesis already tries to mitigate such limitations, there is still the need to design new solutions. First of all, it is necessary to mitigate the problem of data imbalance by proposing alternative data balancing techniques or making the existing ones more sophisticated. Furthermore, given that our results report that combining different sets of metrics leads to a performance improvement, it could be worth exploring even more metrics and also considering the combination between heuristics and machine learning-based techniques (e.g., a classifier could detect an instance as smelly based on the prediction provided by different heuristics). Should even

these alternatives do not lead to acceptable performance, it should be considered a complete change of route, abandoning the idea of solving the problem of code smell detection with machine learning and exploring the suitability of other alternative techniques such as deep learning, search-based optimization, or anomaly detection.

2. **Open Issue 2.** *Cross-project code smell detection.* The ultimate goal of our research on machine learning for code smell detection is to devise a technique having high detection capabilities, being at the same time generalizable. However, the studies proposed in this thesis adopted a within-project model, i.e., they learn from a system to detect code smell instances on the system itself. Clearly, these studies have the only goal of assessing the feasibility of machine learning for code smell detection, therefore, they cannot be applied in practice. Differently, in a cross-project scenario, classifiers are trained on a set of systems to detect code smell instances on another system external to the set used for training. This would make machine learning-based code smell detection applied in practice since it makes it possible to use known validated data to detect code smells on an unseen project. In this regard, we have recently performed a comparison between within- and cross-project approaches for machine learning-based code smell detection with the aim of discovering if a transfer learning approach (cross-project) could bring some benefits in machine learning code smell detection research. Our preliminary results show that there is no statistically significant difference between the two approaches. However, since the transfer learning approach does not perform worse than the "traditional" one, it may be worth to keep investigating on this topic, for instance by accurately tuning the cross-project machine learning pipeline or by defining *ad hoc* software engineering for AI techniques. In our future research agenda we plan to conduct a larger study, involving other machine-learning models that we have not employed so far. Moreover, whenever a new and larger code smell dataset will be available, we plan to replicate this study in order to deepen our knowledge on this topic.

3. **Open Issue 3.** *Defining new metrics to measure test quality and effectiveness.* Our research on technical debt in test code have provided important insights to the research community. Specifically, our findings report that, with respect to what previously stated in the literature, existing metrics describing test code quality and effectiveness have a way lower relation to software code quality. Moreover, as an outcome from a focus group we organized with real software developers, it came out that the set of metrics currently available to describe test code quality and effectiveness is quite limited, therefore novel metrics should be defined to better describe these testing attributes.

13.3 FUTURE RESEARCH DIRECTIONS

In the following, we delineate the future research directions and report some preliminary analyses that we already carried out to face the experienced issues.

13.3.1 *Automatic Test Case Generation 2.0*

To support developers during unit testing activities, the research community has been developing automated mechanisms that aim at generating regression test suites targeting individual units of production code. However, these approaches often fail to generate tests that are well-designed, easily understandable, and maintainable [90].

One of the causes behind the poor maintainability of automatically generated test cases might be connected to the fact that existing approaches do not explicitly follow well-established methodologies or guidelines that suggest taking the problem of *test case granularity* into account [253]. In particular, when developing unit test suites, two levels of granularity should be preserved [117, 222, 253]: first, the creation of tests covering single methods of the production code should be pursued, i.e., *intra-method* testing [253] or basic-unit testing [222]; afterwards, tests exercising the interaction between methods of the class should be developed in order to verify additional

execution paths of the production code that would not be covered otherwise, i.e., *intra-class* testing [253] or unit testing [222]. Besides producing test cases of higher quality, a structured strategy might potentially lead to the generation of tests whose oracle would be easier to be find for developers, as they would be required to check smaller portions of code to identify the expected behavior [18].

Recently, we have faced the problem of granularity in automatic test case generation, advancing the state of the art by pursuing the first steps toward the integration of a systematic strategy within the inner-working of automatic test case generation approaches that might possibly support the production of better and more comprehensible test suites [J07]. We build on top of MOSA [244] to devise an improved technique, coined GRANULAR-MOSA (G-MOSA hereafter), that implements the concepts of intra-method and intra-class testing. Our technique splits the overall search budget in two. In the first half, G-MOSA forces the search-based algorithm to generate intra-method tests by limiting the number of production calls to one. In the second half, the standard MOSA implementation is executed so that the generation can cover an arbitrary number of production methods, hence producing intra-class test cases that exercise the interaction among methods.

Our key findings show that the defined systematic strategy actually allows G-MOSA to create intra-method and intra-class test cases. More importantly, the resulting suites have a lower size per test and a higher maintainability than those generated by MOSA, yet having a statistically similar level of code and mutation coverage.

13.3.2 *From Technical Debt to Social Debt*

One of the aspects that is often taken lightly while developing software relates to the social structure organization, i.e., the structures adopted for communication and collaboration during development phases. However, these aspects have been found to be highly relevant factor for the success of software systems [39, 155, 238, 289, 292]. As an example, Kwan *et al.* [155] showed that the alignment between social and technical structure of the community,

i.e., the so-called socio-technical congruence [39], has an effect on the build success, while Palomba *et al.* [238] found that community-related factors can increase the criticality of source code quality issues. As such, studying software communities does not only represent a way to understand and learn how to reduce social debt, i.e., the unforeseen cost given by a wrong management of the communication/coordination between developers [290], but also to possibly improve the overall quality of the technical products being developed [155, 238].

During the last years, we have started working on these aspects by conducting an empirical study on 25 open-source development communities to evaluate (i) the relationship between community patterns (ie, recurring types of organizational or social structures) and community smells (ie, sub-optimal organizational models within the organizational structure that could indicate the presence of problems related to communication and collaboration), and (ii) the impacts of community patterns on software processes and products [C08, J05]. Our results have shown the presence of some relationships between community pattern and community smell as well as demonstrating that community patterns impacts both products and processes. These findings provide important indications, useful for preventing the occurrence of problems if a community is aware that it is following a certain communication model.

In the future, we plan to go more in-depth on this topic building just-in-time mechanisms to make developers aware of these social aspects in real-time and give them more information for deciding how to implement their pieces of code.

A

LIST OF PUBLICATIONS

The complete list of publications is reported below. The * symbol highlights the publications discussed in this dissertation.

INTERNATIONAL JOURNAL PAPERS

J01 - F. Pecorelli, D. Di Nucci, C. De Roover and A. De Lucia (2020), "A large empirical assessment of the role of data balancing in machine-learning-based code smell detection", Journal of Systems and Software, 110693. *

J02 - F. Pecorelli, D. Di Nucci (2020), "Adaptive Selection of Classifiers for Bug Prediction: A Large- Scale Empirical Analysis of Its Performances and a Benchmark Study", Science of Computer Programming.

J03 - F. Pecorelli, F. Palomba and A. De Lucia (2020), "The Relation of Test-Related Factors to Software Quality: A Case Study on Apache Systems", Empirical Software Engineering. *

J04 - F. Pecorelli, G. Catolino, F. Ferrucci, A. De Lucia and F. Palomba, "Software Testing and Android Applications: A Large-Scale Empirical Study", Empirical Software Engineering. *

J05 - M. De Stefano, E. Iannone, F. Pecorelli, D. A. Tamburri, "Impacts of Software Community Patterns on Process and Product: An Empirical Study", Science of Computer Programming. *

J06 - F. Pecorelli, S. Lujan, V. Lenarduzzi, F. Palomba, and A. De Lucia, "On the Adequacy of Static Analysis Warnings with Respect to Code Smell Prediction", Submitted to the Journal of Empirical Software Engineering. *

J07 - F. Pecorelli, G. Grano, F. Palomba, H. C. Gall, and A. De Lucia, “Toward Granular Automatic Unit Test Case Generation”, Submitted to the Journal of Systems and Software. *

J08 - M. De Stefano, F. Pecorelli, D. Di Nucci, F. Palomba, and A. De Lucia, "Software Engineering for Quantum Programming: How Far Are We?", Submitted to the Journal of Systems and Software.

J09 - X. Li, S. Moreschini, F. Pecorelli, and D. Taibi, "OSSARA: Abandonment Risk Assessment for Embedded Open Source Components", Submitted to the IEEE Software Journal.

J10 - V. Lenarduzzi, F. Pecorelli, N. Saarimäki, S. Lujan, and F. Palomba, "A Critical Comparison on Six Static Analysis Tools: Detection, Agreement, and Precision", Submitted to the Journal of Empirical Software Engineering.

J11 - M. De Stefano, F. Pecorelli, D. Di Nucci, F. Palomba, and A. De Lucia, "Training Code Smell Detection Models with Cross-Project Information: An Empirical Assessment", Submitted to the Journal of Empirical Software Engineering.

INTERNATIONAL CONFERENCE PAPERS

C01 - F. Pecorelli, F. Palomba, D. Di Nucci and A. De Lucia, “Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection”, In Proceedings of the IEEE/ACM International Conference on Program Comprehension (ICPC 2019), Montreal, Canada, 2019. *

C02 - F. Pecorelli, D. Di Nucci, C. De Roover and A. De Lucia, “On the Role of Data Balancing for Machine Learning-Based Code Smell Detection”, In Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE 2019), Tallinn, Estonia, 2019. *

C03 - F. Pecorelli, F. Palomba, F. Khomh and A. De Lucia (2020, May), "Developer-Driven Code Smell Prioritization", In International Conference on Mining Software Repositories. *

C04 - S. Lujan, F. Pecorelli, F. Palomba, A. De Lucia and V. Lenarduzzi (2020, November). "A preliminary study on the adequacy of static analysis warnings with respect to code smell prediction", In Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation (pp. 1-6). *

C05 - M. De Stefano, F. Pecorelli., F. Palomba, and A. De Lucia, (2021, August). Comparing within- and cross-project machine learning algorithms for code smell detection. In Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution (pp. 1-6). *

C06 - F. Pecorelli, (2019, August). Test-related factors and post-release defects: an empirical study. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 1235-1237). *

C07 - F. Pecorelli, G. Catolino, F. Ferrucci, A. De Lucia and F. Palomba (2020, July), "Testing of Mobile Applications in the Wild: A Large-Scale Empirical Study on Android Apps", In Proceedings of the 28th International Conference on Program Comprehension(pp. 296-307).

C08 - M. De Stefano, F. Pecorelli, D. A. Tamburri, F. Palomba and A. De Lucia (2020, June), "Splicing Community Patterns and Smells: A Preliminary Study", In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops(pp. 703-710). *

C09 - M. De Stefano, F. Pecorelli, D. A. Tamburri, F. Palomba, and A. De Lucia (2020, September). Refactoring Recommendations Based on the Optimization of Socio-Technical Congruence. In 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 794-796). *

C10 - S. Lambiase, A. Cupito, F. Pecorelli, A. De Lucia and F. Palomba (2020, July), “Just-In-Time Test Smell Detection and Refactoring: The DARTS Project”, In Proceedings of the 28th International Conference on Program Comprehension (pp. 441-445).

C11 - E. Iannone, F. Pecorelli, D. Di Nucci, F. Palomba and A. De Lucia (2020, July), “Refactoring Android-specific Energy Smells: A Plugin for Android Studio”, In Proceedings of the 28th International Conference on Program Comprehension (pp. 451-455).

C12 - F. Pecorelli, G. Di Lillo, F. Palomba and A. De Lucia (2020, September), “VITRuM: A Plug-In for the Visualization of Test-Related Metrics”, In Proceedings of the International Conference on Advanced Visual Interfaces (pp. 1-3).

C13 - M. De Stefano, M.S. Gambardella, F. Pecorelli, F. Palomba and A. De Lucia (2020, September), “cASpER: A Plug-in for Automated Code Smell Detection and Refactoring”, In Proceedings of the International Conference on Advanced Visual Interfaces (pp. 1-3).

C14 - H. Nguyen, F. Lomio, F. Pecorelli, and V. Lenarduzzi, "PANDORA: Continuous Mining Software Repository and Dataset Generation", Submitted to the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2022).

BIBLIOGRAPHY

[1] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension." In: *Software maintenance and reengineering (CSMR), 2011 15th European conference on*. IEEE. 2011, pp. 181–190.

[2] Konstantinos Adamopoulos, Mark Harman, and Robert M Hierons. "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution." In: *Genetic and evolutionary computation conference*. Springer. 2004, pp. 1338–1349.

[3] Amritanshu Agrawal and Tim Menzies. "Is better data better than better data miners?: on the benefits of tuning smote for defect prediction." In: *Proceedings of the 40th International Conference on Software engineering*. ACM. 2018, pp. 1050–1061.

[4] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. "Mining association rules between sets of items in large databases." In: *Acm sigmod record*. Vol. 22. 2. ACM. 1993, pp. 207–216.

[5] Ahmed Al-Shaaby, Hamoud Aljamaan, and Mohammad Alshayeb. "Bad smell detection using machine learning techniques: a systematic literature review." In: *Arabian Journal for Science and Engineering* 45.4 (2020), pp. 2341–2369.

[6] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2014.

[7] Lucas Amorim, Evandro Costa, Nuno Antunes, Baldoino Fonseca, and Marcio Ribeiro. "Experience report: Evaluating the effectiveness of decision trees for detecting code smells." In: *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*. IEEE. 2015, pp. 261–269.

[8] Marc Andreessen. “Why software is eating the world.” In: *Wall Street Journal* 20.2011 (2011), p. C2.

[9] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. “Using mutation analysis for assessing and comparing testing coverage criteria.” In: *IEEE Transactions on Software Engineering* 32.8 (2006), pp. 608–624.

[10] Maurício Aniche, Gabriele Bavota, Christoph Treude, Marco Aurélio Gerosa, and Arie van Deursen. “Code smells for Model-View-Controller architectures.” In: *Empirical Software Engineering* 23.4 (2018), pp. 2121–2157. ISSN: 1573-7616.

[11] Jean-Yves Antoine, Jeanne Villaneau, and Anaïs Lefeuvre. “Weighted Krippendorff’s alpha is a more reliable metrics for multi-coders ordinal annotations: experimental studies on emotion, opinion and coreference annotation.” In: 2014.

[12] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. “Recovering traceability links between code and documentation.” In: *IEEE transactions on software engineering* 28.10 (2002), pp. 970–983.

[13] Business of Apps. *There are 12 million mobile developers worldwide, and nearly half develop for Android first*.

[14] Roberta Arcoverde, Alessandro Garcia, and Eduardo Figueiredo. “Understanding the longevity of code smells: preliminary results of an explanatory survey.” In: *Proceedings of the 4th Workshop on Refactoring Tools*. ACM. 2011, pp. 33–36.

[15] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. “Machine learning techniques for code smell detection: A systematic literature review and meta-analysis.” In: *Information and Software Technology* 108 (2019), pp. 115–138.

[16] Ricardo Baeza-Yates, Berthier de Araújo Neto Ribeiro, et al. *Modern information retrieval*. New York: ACM Press; Harlow, England: Addison-Wesley, 2011.

[17] Gergő Balogh, Tamás Gergely, Árpád Beszédes, and Tibor Gyimóthy. "Are my unit tests in the right package?" In: *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. 2016, pp. 137–146.

[18] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. "The Oracle Problem in Software Testing: A Survey." In: *IEEE Transactions on Software Engineering* 41.5 (2015), pp. 507–525. ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2372785.

[19] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. "A validation of object-oriented design metrics as quality indicators." In: *IEEE Transactions on software engineering* 22.10 (1996), pp. 751–761.

[20] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. "Automating extract class refactoring: an improved method and its evaluation." In: *Empirical Software Engineering* 19.6 (2014), pp. 1617–1664.

[21] Gabriele Bavota, Andrea De Lucia, and Rocco Oliveto. "Identifying extract class refactoring opportunities using structural and semantic cohesion measures." In: *Journal of Systems and Software* 84.3 (2011), pp. 397–414.

[22] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. "The impact of api change-and fault-proneness on the user ratings of android apps." In: *IEEE Transactions on Software Engineering* 41.4 (2014), pp. 384–407.

[23] Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. "Playing with refactoring: Identifying extract class opportunities through game theory." In: *2010 IEEE International Conference on Software Maintenance*. IEEE. 2010, pp. 1–5.

[24] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. "Methodbook: Recommending move method

refactorings via relational topic models." In: *IEEE Transactions on Software Engineering* 40.7 (2013), pp. 671–694.

[25] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. "Are test smells really harmful? An empirical study." In: *Empirical Software Engineering* 20.4 (2015), pp. 1052–1094.

[26] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. "An empirical analysis of the distribution of unit test smells and their impact on software maintenance." In: *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE. 2012, pp. 56–65.

[27] Moritz Beller, Gousios Georgios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. "Developer Testing in The IDE: Patterns, Beliefs, And Behavior." In: *IEEE Transactions on Software Engineering* (2017).

[28] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. "When, How, and Why Developers (Do Not) Test in Their IDEs." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: ACM, 2015, pp. 179–190. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786843. URL: http://doi.acm.org/10.1145/2786805.2786843.

[29] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization." In: *Journal of Machine Learning Research* 13.Feb (2012), pp. 281–305.

[30] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. "Don't touch my code!: examining the effects of ownership on software quality." In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM. 2011, pp. 4–14.

[31] David Bowes, Tracy Hall, Mark Harman, Yue Jia, Federica Sarro, and Fan Wu. "Mutation-aware fault prediction." In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM. 2016, pp. 330–341.

[32] Leo Breiman. "Random forests." In: *Machine learning* 45.1 (2001), pp. 5–32.

[33] Lionel Briand, Domenico Bianculli, Shiva Nejati, Fabrizio Pastore, and Mehrdad Sabetzadeh. "The case for context-driven software engineering research: Generalizability is overrated." In: *IEEE Software* 34.5 (2017), pp. 72–75.

[34] William J Brown, Raphael C Malveau, Hays W McCormick III, and Thomas J Mowbray. *Refactoring software, architectures, and projects in crisis*. 1998.

[35] David Budgen and Pearl Brereton. "Performing systematic literature reviews in software engineering." In: *Proceedings of the 28th international conference on Software engineering*. 2006, pp. 1051–1052.

[36] Raymond PL Buse and Westley R Weimer. "Learning a metric for code readability." In: *IEEE Transactions on Software Engineering* 36.4 (2010), pp. 546–558.

[37] Xia Cai and Michael R Lyu. "Software reliability modeling with test coverage: Experimentation and measurement with a fault-tolerant software project." In: *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*. IEEE. 2007, pp. 17–26.

[38] Ivan Candela, Gabriele Bavota, Barbara Russo, and Rocco Oliveto. "Using cohesion and coupling for software remodularization: Is it enough?" In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25.3 (2016), p. 24.

[39] Marcelo Cataldo, James D Herbsleb, and Kathleen M Carley. "Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity."

In: *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. 2008, pp. 2–11.

[40] Gemma Catolino. "Does source code quality reflect the ratings of apps?" In: *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. ACM. 2018, pp. 43–44.

[41] Gemma Catolino, Dario Di Nucci, and Filomena Ferrucci. "Cross-Project Just-in-Time Bug Prediction for Mobile Apps: An Empirical Assessment." In: *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE. 2019, pp. 99–110.

[42] Gemma Catolino, Fabio Palomba, Francesca Arcelli Fontana, Andrea De Lucia, Andy Zaidman, and Filomena Ferrucci. "Improving change prediction models with code smell-related information." In: *Empirical Software Engineering* 25.1 (2020).

[43] Gemma Catolino, Fabio Palomba, Andrea De Lucia, Filomena Ferrucci, and Andy Zaidman. "Enhancing change prediction models using developer-related factors." In: *Journal of Systems and Software* 143 (), pp. 14–28.

[44] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. "How the Experience of Development Teams Relates to Assertion Density of Test Classes." In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2019, pp. 223–234.

[45] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. "Not all bugs are the same: Understanding, characterizing, and classifying bug types." In: *Journal of Systems and Software* 152 (2019), pp. 165–181.

[46] Chih-Chung Chang and Chih-Jen Lin. "LIBSVM: a library for support vector machines." In: *ACM transactions on intelligent systems and technology (TIST)* 2.3 (2011), p. 27.

[47] Alexander Chatzigeorgiou and Anastasios Manakos. “Investigating the evolution of bad smells in object-oriented code.” In: *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*. IEEE. 2010, pp. 106–115.

[48] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. “SMOTE: synthetic minority over-sampling technique.” In: *Journal of artificial intelligence research* 16 (2002), pp. 321–357.

[49] M-H Chen, Michael R Lyu, and W Eric Wong. “Effect of code coverage on software reliability measurement.” In: *IEEE Transactions on reliability* 50.2 (2001), pp. 165–170.

[50] Shyam R Chidamber and Chris F Kemerer. “A metrics suite for object oriented design.” In: *IEEE Transactions on software engineering* 20.6 (1994), pp. 476–493.

[51] Wynne W Chin et al. “The partial least squares approach to structural equation modeling.” In: *Modern methods for business research* 295.2 (1998), pp. 295–336.

[52] Garvit Rajesh Choudhary, Sandeep Kumar, Kuldeep Kumar, Alok Mishra, and Cagatay Catal. “Empirical analysis of change metrics for software fault prediction.” In: *Computers & Electrical Engineering* 67 (2018), pp. 15–24.

[53] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. “Automated test input generation for android: Are we there yet?” In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering ASE*. IEEE. 2015, pp. 429–440.

[54] William J Clancey. *Classification problem solving*. Stanford University Stanford, CA, 1984.

[55] Michelle Cleary, Jan Horsfall, and Mark Hayter. “Data collection and sampling in qualitative research: does size matter?” In: *Journal of advanced nursing* (2014), pp. 473–475.

[56] Norman Cliff. “Dominance statistics: Ordinal analyses to answer ordinal questions.” In: *Psychological bulletin* 114.3 (1993), p. 494.

[57] Jacob Cohen. *Statistical power analysis*. Vol. 1. 3. Sage Publications Sage CA: Los Angeles, CA, 1992, pp. 98–101.

[58] William W. Cohen. "Fast Effective Rule Induction." In: *Twelfth International Conference on Machine Learning*. Morgan Kaufmann, 1995, pp. 115–123.

[59] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. "A framework for evaluating the results of the szz approach for identifying bug-introducing changes." In: *IEEE Transactions on Software Engineering* 43.7 (2017), pp. 641–657.

[60] Steve Counsell, Stephen Swift, and Jason Crampton. "The interpretation and utility of three cohesion metrics for object-oriented design." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 15.2 (2006), pp. 123–149.

[61] John W Creswell. "Mixed-method research: Introduction and application." In: *Handbook of educational policy*. Elsevier, 1999, pp. 455–472.

[62] Luis Cruz, Rui Abreu, and David Lo. "To the attention of mobile software developers: guess what, test your app!" In: *Empirical Software Engineering* (2019), pp. 1–31.

[63] Ward Cunningham. "The WyCash portfolio management system." In: *ACM SIGPLAN OOPS Messenger* 4.2 (1993), pp. 29–30.

[64] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. "On the impact of design flaws on software defects." In: *2010 10th International Conference on Quality Software*. IEEE. 2010, pp. 23–31.

[65] Marco D'Ambros, Michele Lanza, and Romain Robbes. "On the relationship between change coupling and software defects." In: *2009 16th Working Conference on Reverse Engineering*. IEEE. 2009, pp. 135–144.

[66] Lars-Ola Damm, Lars Lundberg, and Claes Wohlin. "Faults-slip-through—a concept for measuring the efficiency of the test process." In: *Software Process: Improvement and Practice* 11.1 (2006), pp. 47–59.

[67] Teerath Das, Massimiliano Di Penta, and Ivano Malavolta. "A quantitative and qualitative investigation of performance-related commits in android apps." In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2016, pp. 443–447.

[68] Dario Di Nucci, Fabio Palomba, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia. "A developer centered bug prediction model." In: *IEEE Transactions on Software Engineering* 44.1 (2018), pp. 5–24.

[69] Dario Di Nucci, Fabio Palomba, Rocco Oliveto, and Andrea De Lucia. "Dynamic Selection of Classifiers in Bug Prediction: An Adaptive Method." In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 1.3 (2017), pp. 202–212.

[70] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. "Software-based energy profiling of android apps: Simple, efficient and reliable?" In: *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE. 2017, pp. 103–114.

[71] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. "Detecting code smells using machine learning techniques: are we there yet?" In: *25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER2018): REproducibility Studies and NEgative Results (RENE) Track*. Institute of Electrical and Electronics Engineers (IEEE). 2018.

[72] David J Dittman, Taghi M Khoshgoftaar, Randall Wald, and Amri Napolitano. "Comparison of data sampling approaches for imbalanced bioinformatics data." In: *The twenty-seventh international FLAIRS conference*. 2014.

[73] Norman R Draper and Harry Smith. *Applied regression analysis*. Vol. 326. John Wiley & Sons, 2014.

[74] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. "Understanding Flaky Tests: The Developer's Perspective." In: (2019), to appear.

[75] Letha H Etzkorn, Sampson E Gholston, Julie L Fortune, Cara E Stein, Dawn Utley, Phillip A Farrington, and Glenn W Cox. "A comparison of cohesion metrics for object-oriented systems." In: *Information and Software Technology* 46.10 (2004), pp. 677–687.

[76] Robert Feldt. "Do system test cases grow old?" In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE. 2014, pp. 343–352.

[77] Robert Feldt, Richard Torkar, Tony Gorschek, and Wasif Afzal. "Searching for cognitively diverse tests: Towards universal test diversity metrics." In: *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*. IEEE. 2008, pp. 178–186.

[78] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. "A review-based comparative study of bad smell detection tools." In: *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. ACM. 2016, p. 18.

[79] Leonardo Fernandes, Márcio Ribeiro, Luiz Carvalho, Rohit Gheyi, Melina Mongiovi, André Santos, Ana Cavalcanti, Fabiano Ferrari, and José Carlos Maldonado. "Avoiding useless mutants." In: *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 2017, pp. 187–198.

[80] Michael Fischer, Martin Pinzger, and Harald Gall. "Populating a release history database from version control and bug tracking systems." In: *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. IEEE. 2003, pp. 23–32.

[81] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. "Identification and application of extract class refactorings in object-oriented systems." In: *Journal of Systems and Software* 85.10 (2012), pp. 2241–2260.

[82] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. "Automatic detection of bad smells in code: An experimental assessment." In: *Journal of Object Technology* 11.2 (2012), pp. 5–1.

[83] Francesca Arcelli Fontana, Jens Dietrich, Bartosz Walter, Aiko Yamashita, and Marco Zanoni. "Antipattern and code smell false positives: Preliminary conceptualization and classification." In: *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*. Vol. 1. IEEE. 2016, pp. 609–613.

[84] Francesca Arcelli Fontana, Vincenzo Ferme, and Marco Zanoni. "Filtering code smells detection results." In: *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press. 2015, pp. 803–804.

[85] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. "Comparing and experimenting machine learning techniques for code smell detection." In: *Empirical Software Engineering* 21.3 (2016), pp. 1143–1191.

[86] Francesca Arcelli Fontana and Marco Zanoni. "Code smell severity classification using machine learning techniques." In: *Knowledge-Based Systems* 128 (2017), pp. 43–58.

[87] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V Mantyla. "Code smell detection: Towards a machine learning-based approach." In: *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE. 2013, pp. 396–399.

[88] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[89] Gordon Fraser and Andrea 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*. ESEC/FSE '11. Szeged, Hungary: ACM, 2011, pp. 416–419. ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025179. URL: http://doi.acm.org/10.1145/2025113.2025179.

[90] Gordon Fraser and Andrea Arcuri. "Whole Test Suite Generation." In: *IEEE Trans. Softw. Eng.* 39.2 (Feb. 2013), pp. 276–291. ISSN: 0098-5589. DOI: 10.1109/TSE.2012.14. URL: http://dx.doi.org/10.1109/TSE.2012.14.

[91] Enrico Fregnan, Tobias Baum, Fabio Palomba, and Alberto Bacchelli. "A survey on software coupling relations and tools." In: *Information and Software Technology* (2018).

[92] Mikel Galar, Alberto Fernandez, Edurne Barrenechea, Humberto Bustince, and Francisco Herrera. "A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches." In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.4 (2011), pp. 463–484.

[93] Jerry Gao, Wei-Tek Tsai, Ray Paul, Xiaoying Bai, and Tadahiro Uehara. "Mobile Testing-as-a-Service (MTaaS)–Infrastructures, Issues, Solutions and Needs." In: *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*. IEEE. 2014, pp. 158–167.

[94] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. "The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature." In: *Proceedings of the 20th international conference on evaluation and assessment in software engineering*. 2016, pp. 1–6.

[95] Vahid Garousi and Barış Küçük. "Smells in software test code: A survey of knowledge in industry and academia." In: *Journal of systems and software* 138 (2018), pp. 52–81.

[96] Gartner. *Information technology (IT) spending on enterprise software worldwide, from 2009 to 2022 (in billion U.S. dollars)*. Oct. 2021.

[97] Franz-Xaver Geiger and Ivano Malavolta. "Datasets of Android Applications: a Literature Review." In: *arXiv preprint arXiv:1809.10069* (2018).

[98] Franz-Xaver Geiger, Ivano Malavolta, Luca Pascarella, Fabio Palomba, Dario Di Nucci, and Alberto Bacchelli. "A graph-based dataset of commit history of real-world android apps." In: *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM. 2018, pp. 30–33.

[99] Peter Gilbert, Byung-Gon Chun, Landon P Cox, and Jaeyeon Jung. "Vision: automated security validation of mobile apps at app markets." In: *Proceedings of the second international workshop on Mobile cloud computing and services*. ACM. 2011, pp. 21–26.

[100] Rahul Gopinath, Carlos Jensen, and Alex Groce. "Code coverage for suite evaluation by developers." In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 72–82.

[101] Giovanni Grano, Adelina Ciurumelea, Sebastiano Panichella, Fabio Palomba, and Harald C Gall. "Exploring the integration of user feedback in automated testing of android applications." In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2018, pp. 72–83.

[102] Giovanni Grano, Cristian De Iaco, Fabio Palomba, and Harald C Gall. "Pizza versus Pinsa: On the Perception and Measurability of Unit Test Code Quality." In: (2020), to appear.

[103] Giovanni Grano, Fabio Palomba, Dario Di Nucci, Andrea De Lucia, and Harald C Gall. "Scented since the beginning: On the diffuseness of test smells in automatically generated test code." In: *Journal of Systems and Software* 156 (2019), pp. 312–327.

[104] Giovanni Grano, Fabio Palomba, and Harald C Gall. "Lightweight Assessment of Test-Case Effectiveness using Source-Code-Quality Indicators." In: *IEEE Transactions on Software Engineering* (2019).

[105] Giovanni Grano, Simone Scalabrino, Rocco Oliveto, and Harald Gall. "An Empirical Investigation on the Readability of Manual and Generated Test Cases." In: *Proceedings of the 26th International Conference on Program Comprehension, ICPC*. 2018.

[106] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. "Predicting fault incidence using software change history." In: *IEEE Transactions on software engineering* 26.7 (2000), pp. 653–661.

[107] Michaela Greiler, Arie Van Deursen, and Margaret-Anne Storey. "Automated detection of test fixture strategies and smells." In: *Software Testing, Verification and Validation (ICST)*. 2013, pp. 322–331.

[108] Robert J. Grissom and John J. Kim. *Effect sizes for research: A broad practical approach*. 2nd Edition. Lawrence Earlbaum Associates, 2005.

[109] Gui Gui and Paul D Scott. "Coupling and cohesion measures for evaluation of component reusability." In: *Proceedings of the 2006 international workshop on Mining software repositories*. 2006, pp. 18–21.

[110] David Gunning. "Explainable artificial intelligence (xai)." In: *Defense Advanced Research Projects Agency (DARPA), nd Web* 2 (2017).

[111] Sonia Haiduc, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Andrian Marcus. "Automatic query performance assessment during the retrieval of software artifacts." In: *Proceedings of the 27th IEEE/ACM international conference on Automated Software Engineering*. ACM. 2012, pp. 90–99.

[112] Ulrich Halekoh, Søren Højsgaard, Jun Yan, et al. "The R package geepack for generalized estimating equations." In: *Journal of Statistical Software* 15.2 (2006), pp. 1–11.

[113] Mark A. Hall. *Correlation-based feature selection for machine learning*. Tech. rep. 1998.

[114] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. "The WEKA data mining software: an update." In: *ACM SIGKDD explorations newsletter* 11.1 (2009), pp. 10–18.

[115] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. "Developing fault-prediction models: What the research can show industry." In: *IEEE software* 28.6 (2011), pp. 96–99.

[116] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. "Some code smells have a significant but small effect on faults." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23.4 (2014), p. 33.

[117] Mary Jean Harrold, John D McGregor, and Kevin J Fitzpatrick. "Incremental testing of object-oriented class structures." In: *Proceedings of the 14th international conference on Software engineering*. 1992, pp. 68–80.

[118] Salima Hassaine, Foutse Khomh, Yann-Gaël Guéhéneuc, and Sylvie Hamel. "IDS: An immune-inspired approach for the detection of software design smells." In: *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*. IEEE. 2010, pp. 343–348.

[119] Ahmed E Hassan. "Predicting faults using the complexity of code changes." In: *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE. 2009, pp. 78–88.

[120] Andrew F Hayes. "Beyond Baron and Kenny: Statistical mediation analysis in the new millennium." In: *Communication monographs* 76.4 (2009), pp. 408–420.

[121] Brian Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1995.

[122] Brian Henderson-Sellers, Larry L Constantine, and Ian M Graham. "Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design)." In: *Object oriented systems* 3.3 (1996), pp. 143–158.

[123] Abram Hindle, Alex Wilson, Kent Rasmussen, E Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. "Greenminer: A hardware based mining software repositories software energy consumption framework." In: *Proceedings of the 11th working conference on mining software repositories*. 2014, pp. 12–21.

[124] Emanuele Iannone, Fabiano Pecorelli, Dario Di Nucci, Fabio Palomba, and Andrea De Lucia. "Refactoring Android-specific Energy Smells: A Plugin for Android Studio." In: *Proceedings of the 28th International Conference on Program Comprehension*. 2020, pp. 451–455.

[125] Chadni Islam, Muhammad Ali Babar, and Surya Nepal. "A Multi-Vocal Review of Security Orchestration." In: *ACM Computing Surveys (CSUR)* 52.2 (2019), pp. 1–45.

[126] George H John and Pat Langley. "Estimating continuous distributions in Bayesian classifiers." In: *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc. 1995, pp. 338–345.

[127] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. "Why don't software developers use static analysis tools to find bugs?" In: *35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 672–681.

[128] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. "Real challenges in mobile app development." In: *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE. 2013, pp. 15–24.

[129] Suhas Kabinna, Weiyi Shang, Cor-Paul Bezemer, and Ahmed E Hassan. "Examining the stability of logging statements." In: *Software*

Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd Int'l Conf. on. Vol. 1. IEEE. 2016, pp. 326–337.

[130] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. "A large-scale empirical study of just-in-time quality assurance." In: *IEEE Transactions on Software Engineering* 39.6 (2013), pp. 757–773. ISSN: 0098-5589. DOI: 10.1109/TSE.2012.70.

[131] Amandeep Kaur, Sushma Jain, and Shivani Goel. "A support vector machine based approach for code smell detection." In: *2017 International Conference on Machine Learning and Data Science (MLDS)*. IEEE. 2017, pp. 9–14.

[132] Amandeep Kaur, Sushma Jain, Shivani Goel, and Gaurav Dhiman. "A review on machine-learning based code smell detection techniques in object-oriented software system (s)." In: *Recent Advances in Electrical & Electronic Engineering (Formerly Recent Patents on Electrical & Electronic Engineering)* 14.3 (2021), pp. 290–303.

[133] Staffs Keele et al. *Guidelines for performing systematic literature reviews in software engineering*. Tech. rep. Technical report, Ver. 2.3 EBSE Technical Report. EBSE, 2007.

[134] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E Hassan. "What do mobile app users complain about?" In: *IEEE Software* 32.3 (2014), pp. 70–77.

[135] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. "An Exploratory Study of the Impact of Code Smells on Software Change-proneness." In: *Proceedings of the 2009 16th Working Conference on Reverse Engineering*. WCRE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 75–84. ISBN: 978-0-7695-3867-9. DOI: 10.1109/WCRE.2009.28.

[136] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. "An exploratory study of the impact of antipatterns on class change- and fault-proneness." In: *Empirical Software Engineering* 17.3 (2012), pp. 243–275.

[137] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. "An exploratory study of the impact of antipatterns on class change-and fault-proneness." In: *Empirical Software Engineering* 17.3 (2012), pp. 243–275.

[138] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. "A bayesian approach for the detection of code and design smells." In: *International Conference on Quality Software*. IEEE. 2009, pp. 305–314.

[139] Foutse Khomh, Stephane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. "BDTEX: A GQM-based Bayesian approach for the detection of antipatterns." In: *Journal of Systems and Software* 84.4 (2011), pp. 559–572.

[140] Heejin Kim, Byoungju Choi, and W Eric Wong. "Performance testing of mobile applications at the unit test level." In: *2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement*. IEEE. 2009, pp. 171–180.

[141] Sunghun Kim, E. James Whitehead, and Yi Zhang. "Classifying software changes: Clean or buggy?" In: *IEEE Transactions on Software Engineering* 34.2 (2008), pp. 181–196. ISSN: 00985589. DOI: 10.1109/TSE.2007.70773.

[142] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. "Predicting faults from cached history." In: *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society. 2007, pp. 489–498.

[143] Barbara Kitchenham. "Procedures for performing systematic reviews." In: *Keele, UK, Keele University* 33.2004 (2004), pp. 1–26.

[144] Patrick Knab, Martin Pinzger, and Abraham Bernstein. "Predicting defect densities in source code files with decision tree learners." In: *Proceedings of the 2006 international workshop on Mining software repositories*. ACM. 2006, pp. 119–125.

[145] Pavneet Singh Kochhar, David Lo, Julia Lawall, and Nachiappan Nagappan. "Code coverage and postrelease defects: A large-scale study on open source projects." In: *IEEE Transactions on Reliability* 66.4 (2017), pp. 1213–1228.

[146] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. "Understanding the test automation culture of app developers." In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation ICST*. IEEE. 2015, pp. 1–10.

[147] Ron Kohavi et al. "A study of cross-validation and bootstrap for accuracy estimation and model selection." In: *Ijcai*. Vol. 14. 2. Montreal, Canada. 1995, pp. 1137–1145.

[148] Alex J Koning, Philip Hans Franses, Michele Hibon, and Herman O Stekler. "The M3 competition: Statistical tests of the results." In: *International Journal of Forecasting* 21.3 (2005), pp. 397–409.

[149] A Güneş Koru, Dongsong Zhang, Khaled El Emam, and Hongfang Liu. "An investigation into the functional form of the size-defect relationship for software modules." In: *IEEE Transactions on Software Engineering* 35.2 (2009), pp. 293–304.

[150] Sotiris Kotsiantis, Dimitris Kanellopoulos, Panayiotis Pintelas, et al. "Handling imbalanced datasets: A review." In: *GESTS International Transactions on Computer Science and Engineering* 30.1 (2006), pp. 25–36.

[151] Jochen Kreimer. "Adaptive detection of design flaws." In: *Electronic Notes in Theoretical Computer Science* 141.4 (2005), pp. 117–136.

[152] Klaus Krippendorff. *Content analysis: An introduction to its methodology*. Sage publications, 2018.

[153] Daniel E Krutz, Mehdi Mirakhorli, Samuel A Malachowsky, Andres Ruiz, Jacob Peterson, Andrew Filipski, and Jared Smith. "A dataset of open-source Android applications." In: *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press. 2015, pp. 522–525.

[154] Gunnar Kudrjavets, Nachiappan Nagappan, and Thomas Ball. "Assessing the relationship between software assertions and faults: An empirical investigation." In: *2006 17th International Symposium on Software Reliability Engineering*. IEEE. 2006, pp. 204–212.

[155] Irwin Kwan, Adrian Schroter, and Daniela Damian. "Does Socio-Technical Congruence Have an Effect on Software Build Success? A Study of Coordination in a Software Project." In: *IEEE Trans. Softw. Eng.* 37.3 (May 2011), pp. 307–324. ISSN: 0098-5589. DOI: 10.1109/TSE.2011.29.

[156] Christoph Laaber and Philipp Leitner. "An evaluation of open-source software microbenchmark suites for continuous performance assessment." In: *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM. 2018, pp. 119–130.

[157] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[158] Niklas Leicht, Ivo Blohm, and Jan Marco Leimeister. "Leveraging the power of the crowd for software testing." In: *IEEE Software* 34.2 (2017), pp. 62–69.

[159] V. Lenarduzzi, A. Sillitti, and D. Taibi. "A Survey on Code Analysis Tools for Software Maintenance Prediction." In: *6th International Conference in Software Engineering for Defence Applications*. Springer International Publishing, 2020, pp. 165–175.

[160] Heng Li, Weiyi Shang, Ying Zou, and Ahmed E Hassan. "Towards just-in-time suggestions for log changes." In: *Empirical Software Engineering* (2016), pp. 1–35.

[161] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage." In: *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*. 2009, pp. 220–229.

[162] Wei Li and Sallie Henry. “Maintenance metrics for the object oriented paradigm.” In: *[1993] Proceedings First International Software Metrics Symposium*. IEEE. 1993, pp. 52–60.

[163] Rensis Likert. “A technique for the measurement of attitudes.” In: *Archives of psychology* (1932).

[164] Jun-Wei Lin, Navid Salehnamadi, and Sam Malek. “Test automation in open-source android apps: A large-scale empirical study.” In: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2020, pp. 1078–1089.

[165] Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. “How do developers test android applications?” In: *2017 IEEE International Conference on Software Maintenance and Evolution ICSME*. IEEE. 2017, pp. 613–622.

[166] Mario Linares-Vásquez, Sam Klock, Collin McMillan, Aminata Sabané, Denys Poshyvanyk, and Yann-Gaël Guéhéneuc. “Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in Java mobile apps.” In: *Proceedings of the 22nd International Conference on Program Comprehension*. 2014, pp. 232–243.

[167] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. “Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing.” In: *2017 IEEE International Conference on Software Maintenance and Evolution ICSME*. IEEE. 2017, pp. 399–410.

[168] Charles X Ling and Chenghui Li. “Data mining for direct marketing: Problems and solutions.” In: *Kdd*. Vol. 98. 1998, pp. 73–79.

[169] Huan Liu and Hiroshi Motoda. *Feature selection for knowledge discovery and data mining*. Vol. 454. Springer Science & Business Media, 2012.

[170] Angela Lozano, Michel Wermelinger, and Bashar Nuseibeh. "Assessing the impact of bad smells using historical information." In: *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*. ACM. 2007, pp. 31–34.

[171] Zeeger Lubsen, Andy Zaidman, and Martin Pinzger. "Using association rules to study the co-evolution of production & test code." In: *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*. IEEE. 2009, pp. 151–154.

[172] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. "An empirical analysis of flaky tests." In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 643–653.

[173] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. "Dynodroid: An input generation system for android apps." In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM. 2013, pp. 224–234.

[174] Lech Madeyski and Marian Jureczko. "Which process metrics can significantly improve defect prediction models? An empirical study." In: *Software Quality Journal* 23.3 (2015), pp. 393–422.

[175] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation." In: *IEEE Transactions on Software Engineering* 40.1 (2013), pp. 23–42.

[176] Zaheed Mahmood, David Bowes, Peter CR Lane, and Tracy Hall. "What is the impact of imbalance on software defect prediction performance?" In: *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM. 2015, p. 4.

[177] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabane, Yann-Gael Gueheneuc, and Esma Aimeur. "SMURF: A SVM-based incremental anti-pattern detection approach." In: *Reverse engineering (WCRE), 2012 19th working conference on*. IEEE. 2012, pp. 466–475.

[178] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Esma Aïmeur. "Support vector machines for anti-pattern detection." In: *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*. IEEE. 2012, pp. 278–281.

[179] Usman Mansoor, Marouane Kessentini, Bruce R Maxim, and Kalyanmoy Deb. "Multi-objective code-smells detection using good and bad design examples." In: *Software Quality Journal* 25.2 (2017), pp. 529–552.

[180] Mika V Mäntylä, Bram Adams, Foutse Khomh, Emelie Engström, and Kai Petersen. "On rapid releases and software testing: a case study and a semi-systematic literature review." In: *Empirical Software Engineering* 20.5 (2015), pp. 1384–1425.

[181] Mika V Mäntylä and Casper Lassenius. "Subjective evaluation of software evolvability using code smells: An empirical study." In: *Empirical Software Engineering* 11.3 (2006), pp. 395–431.

[182] Ke Mao, Mark Harman, and Yue Jia. "Sapienz: Multi-objective automated testing for Android applications." In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM. 2016, pp. 94–105.

[183] Andrian Marcus and Denys Poshyvanyk. "The conceptual cohesion of classes." In: *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE. 2005, pp. 133–142.

[184] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. "Using the conceptual cohesion of classes for fault prediction in object-oriented systems." In: *IEEE Transactions on Software Engineering* 34.2 (2008), pp. 287–300.

[185] Brian Marick et al. “How to misuse code coverage.” In: *Proceedings of the 16th Interational Conference on Testing Computer Software*. 1999, pp. 16–18.

[186] Radu Marinescu. “Detection strategies: Metrics-based rules for detecting design flaws.” In: *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. IEEE. 2004, pp. 350–359.

[187] Radu Marinescu. “Assessing technical debt by identifying design flaws in software systems.” In: *IBM Journal of Research and Development* 56.5 (2012), pp. 9–1.

[188] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. “A survey of app store analysis for software engineering.” In: *IEEE transactions on software engineering* 43.9 (2016), pp. 817–847.

[189] Katsuhisa Maruyama. “Automated method-extraction refactoring by using block-based slicing.” In: *Proceedings of the 2001 symposium on Software reusability: putting software reuse in context*. 2001, pp. 31–40.

[190] Bruno Gois Mateus and Matias Martinez. “An empirical study on quality of Android applications written in Kotlin language.” In: *Empirical Software Engineering* 24.6 (2019), pp. 3356–3393.

[191] Thomas J McCabe. “A complexity measure.” In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.

[192] Stuart McIlroy, Nasir Ali, and Ahmed E Hassan. “Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store.” In: *Empirical Software Engineering* 21.3 (2016), pp. 1346–1370.

[193] Tim Menzies, Andrew Butcher, David Cok, Andrian Marcus, Lucas Layman, Forrest Shull, Burak Turhan, and Thomas Zimmermann. “Local versus global lessons for defect prediction and effort estimation.” In: *IEEE Transactions on software engineering* 39.6 (2013), pp. 822–834.

[194] Ali Mesbah and Mukul R Prasad. "Automated cross-browser compatibility testing." In: *Proceedings of the 33rd International Conference on Software Engineering*. ACM. 2011, pp. 561–570.

[195] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[196] Roberto Minelli and Michele Lanza. "Software Analytics for Mobile Applications–Insights & Lessons Learned." In: *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE. 2013, pp. 144–153.

[197] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. "DECOR: A method for the specification and detection of code and design smells." In: *IEEE Trans. on Software Engineering* 36.1 (2010), pp. 20–36.

[198] Israel J Mojica, Bram Adams, Meiyappan Nagappan, Steffen Dienst, Thorsten Berger, and Ahmed E Hassan. "A large-scale empirical study on software reuse in mobile apps." In: *IEEE software* 31.2 (2013), pp. 78–86.

[199] Leon Moonen. "Generating Robust Parsers Using Island Grammars." In: *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001*. 2001, p. 13.

[200] Rodrigo Morales, Ruben Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. "Anti-patterns and the energy efficiency of Android applications." In: *arXiv preprint arXiv:1610.05711* (2016).

[201] Catriona M Morrison. "Interpret with caution: multicollinearity in multiple regression of cognitive data." In: *Perceptual and motor skills* 97.1 (2003), pp. 80–82.

[202] Henry Muccini, Antonio Di Francesco, and Patrizio Esposito. "Software testing of mobile applications: Challenges and future research directions." In: *Proceedings of the 7th International Workshop on Automation of Software Test*. IEEE Press. 2012, pp. 29–35.

[203] Matthew James Munro. “Product metrics for automatic identification of" bad smell" design problems in java source-code.” In: *11th IEEE International Software Metrics Symposium (METRICS’05)*. IEEE. 2005, pp. 15–15.

[204] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[205] Meiyappan Nagappan and Emad Shihab. “Future trends in software engineering research for mobile apps.” In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 5. IEEE. 2016, pp. 21–32.

[206] Nachiappan Nagappan and Thomas Ball. “Use of relative code churn measures to predict system defect density.” In: *Proceedings of the 27th international conference on Software engineering*. ACM. 2005, pp. 284–292.

[207] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. “Mining metrics to predict component failures.” In: *Proceedings of the 28th international conference on Software engineering*. ACM. 2006, pp. 452–461.

[208] Nachiappan Nagappan, E Michael Maximilien, Thirumalesh Bhat, and Laurie Williams. “Realizing quality improvement through test driven development: results and experiences of four industrial teams.” In: *Empirical Software Engineering* 13.3 (2008), pp. 289–302.

[209] Nachiappan Nagappan, Laurie Williams, Mladen Vouk, and Jason Osborne. “Early estimation of software quality using in-process testing metrics: a controlled case study.” In: *ACM SIGSOFT Software Engineering Notes*. Vol. 30. 4. ACM. 2005, pp. 1–7.

[210] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. “Change bursts as defect predictors.” In: *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE. 2010, pp. 309–318.

[211] John F Nash et al. “Equilibrium points in n-person games.” In: *Proceedings of the national academy of sciences* 36.1 (1950), pp. 48–49.

[212] Maleknaz Nayebi, Bram Adams, and Guenther Ruhe. “Release Practices for Mobile Apps–What do Users and Developers Think?” In: *2016 ieee 23rd international conference on software analysis, evolution, and reengineering (saner)*. Vol. 1. IEEE. 2016, pp. 552–562.

[213] John Ashworth Nelder and Robert WM Wedderburn. “Generalized linear models.” In: *Journal of the Royal Statistical Society: Series A (General)* 135.3 (1972), pp. 370–384.

[214] Peter Nemenyi. “Distribution-free multiple comparisons.” In: *Biometrics*. Vol. 18. 2. International Biometric Soc 1441 I ST NW SUITE 700, WASHINGTON DC 20005-2210. 1962, p. 263.

[215] John Neter, Michael H Kutner, Christopher J Nachtsheim, and William Wasserman. *Applied linear statistical models*. Vol. 4. Irwin Chicago, 1996.

[216] Raymond S Nickerson. “Confirmation bias: A ubiquitous phenomenon in many guises.” In: *Review of general psychology* 2.2 (1998), pp. 175–220.

[217] Robert M O?brien. “A caution regarding rules of thumb for variance inflation factors.” In: *Quality & Quantity* 41.5 (2007), pp. 673–690.

[218] A Jefferson Offutt and Roland H Untch. “Mutation 2000: Uniting the orthogonal.” In: *Mutation testing for the new century*. Springer, 2001, pp. 34–44.

[219] Steffen Olbrich, Daniela S Cruzes, Victor Basili, and Nico Zazworka. “The evolution and impact of code smells: A case study of two open source systems.” In: *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*. IEEE. 2009, pp. 390–400.

[220] Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. "On the equivalence of information retrieval methods for automated traceability link recovery." In: *2010 IEEE 18th International Conference on Program Comprehension*. IEEE. 2010, pp. 68–71.

[221] Rocco Oliveto, Foutse Khomh, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. "Numerical signatures of antipatterns: An approach based on b-splines." In: *Software maintenance and reengineering (CSMR), 2010 14th European Conference on*. IEEE. 2010, pp. 248–251.

[222] Alessandro Orso and Sergio Silva. "Open Issues and Research Directions in Object-Oriented Testing." In: *Proceedings of the 4th International Conference on" Achieving Quality in Software: Software Quality in the Communication Society"(AQUIS'98)*. 1998.

[223] Juliana Padilha, Juliana Pereira, Eduardo Figueiredo, Jussara Almeida, Alessandro Garcia, and Cláudio Sant'Anna. "On the effectiveness of concern metrics to detect code smells: an empirical study." In: *International Conference on Advanced Information Systems Engineering*. Springer. 2014, pp. 656–671.

[224] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. "On the Diffuseness and the Impact on Maintainability of Code Smells: A Large Scale Empirical Study." In: *Empirical Software Engineering* (2017), to appear.

[225] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. "A large-scale empirical study on the lifecycle of code smell co-occurrences." In: *Information and Software Technology* 99 (2018), pp. 1–10.

[226] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. "On the diffuseness and the impact on maintainability of code smells: a large scale

empirical investigation." In: *Empirical Software Engineering* 23.3 (2018), pp. 1188–1221.

[227] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. "Do they really smell bad? a study on developers' perception of bad code smells." In: *Software maintenance and evolution (ICSME), 2014 IEEE international conference on*. IEEE. 2014, pp. 101–110.

[228] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. "Mining version histories for detecting code smells." In: *IEEE Transactions on Software Engineering* 41.5 (2015), pp. 462–489.

[229] Fabio Palomba, Andrea De Lucia, Gabriele Bavota, and Rocco Oliveto. "Anti-pattern detection: Methods, challenges, and open issues." In: *Advances in Computers*. Vol. 95. Elsevier, 2014, pp. 201–238.

[230] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea 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*. ACM. 2016, pp. 5–14.

[231] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. "On the impact of code smells on the energy consumption of mobile applications." In: *Information and Software Technology* 105 (2019), pp. 43–55.

[232] Fabio Palomba, Dario Di Nucci, Michele Tufano, Gabriele Bavota, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. "Landfill: An open dataset of code smells with public evaluation." In: *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*. IEEE. 2015, pp. 482–485.

[233] Fabio Palomba, Mario Linares-Vásquez, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. "Crowdsourcing user reviews to support the evolution

of mobile apps." In: *Journal of Systems and Software* 137 (2018), pp. 143–162.

[234] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. "A textual-based technique for smell detection." In: *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE. 2016, pp. 1–10.

[235] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. "Automatic test case generation: What if test code quality matters?" In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM. 2016, pp. 130–141.

[236] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. "The scent of a smell: An extensive comparison between textual and structural smells." In: *IEEE Transactions on Software Engineering* 44.10 (2018), pp. 977–1000.

[237] Fabio Palomba, Pasquale Salza, Adelina Ciurumelea, Sebastiano Panichella, Harald Gall, Filomena Ferrucci, and Andrea De Lucia. "Recommending and localizing change requests for mobile apps based on user reviews." In: *Proceedings of the 39th international conference on software engineering*. IEEE Press. 2017, pp. 106–117.

[238] Fabio Palomba, Damian Andrew Andrew Tamburri, Francesca Arcelli Fontana, Rocco Oliveto, Andy Zaidman, and Alexander Serebrenik. "Beyond technical aspects: How do community smells influence the intensity of code smells?" In: *IEEE transactions on software engineering* (2018).

[239] Fabio Palomba and Andy Zaidman. "Does refactoring of test smells induce fixing flaky tests?" In: *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE. 2017, pp. 1–12.

[240] Fabio Palomba and Andy Zaidman. "The Smell of Fear: On the Relation between Test Smells and Flaky Tests." In: *Empirical Software Engineering Journal* (2019), in press.

[241] Fabio Palomba, Andy Zaidman, and Andrea De Lucia. "Automatic test smell detection using information retrieval techniques." In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2018, pp. 311–322.

[242] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. "An exploratory study on the relationship between changes and refactoring." In: *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on*. IEEE. 2017, pp. 176–185.

[243] Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia, and Rocco Oliveto. "Toward a Smell-aware Bug Prediction Model." In: *IEEE Transactions on Software Engineering* (2017).

[244] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. "Reformulating Branch Coverage as a Many-Objective Optimization Problem." In: *ICST*. IEEE Computer Society, 2015, pp. 1–10.

[245] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. "The impact of test case summaries on bug fixing performance: an empirical investigation." In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 2016, pp. 547–558. DOI: 10.1145/2884781.2884847. URL: http://doi.acm.org/10.1145/2884781.2884847.

[246] Jevgenija Pantiuchina, Gabriele Bavota, Michele Tufano, and Denys Poshyvanyk. "Towards just-in-time refactoring recommenders." In: *Proceedings of the 26th Conference on Program Comprehension*. 2018, pp. 312–315.

[247] Reza Meimandi Parizi, Sai Peck Lee, and Mohammad Dabbagh. "Achievements and challenges in state-of-the-art software traceability between test and code artifacts." In: *IEEE Transactions on Reliability* 63.4 (2014), pp. 913–926.

[248] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. "Fine-grained just-in-time defect prediction." In: *Journal of Systems and Software* 150 (2019), pp. 22–36.

[249] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. “A systematic literature review on bad smells—5 W’s: which, when, what, who, where.” In: *IEEE Transactions on Software Engineering* (2018).

[250] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. “On the distribution of test smells in open source Android applications: an exploratory study.” In: *CASCON*. 2019, pp. 193–202.

[251] Anthony Peruma, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. “An Exploratory Study on the Refactoring of Unit Test Files in Android Applications.” In: *Conference on Software Engineering Workshops (ICSEW’20)*. 2020.

[252] Ralph Peters and Andy Zaidman. “Evaluating the lifespan of code smells using software repository mining.” In: *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE. 2012, pp. 411–416.

[253] Mauro Pezzè and Michal Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[254] Raphael Pham, Stephan Kiesling, Olga Liskin, Leif Singer, and Kurt Schneider. “Enablers, inhibitors, and perceptions of testing in novice software teams.” In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 30–40.

[255] Nick Pidgeon and Karen Henwood. *Grounded theory*. na, 2004.

[256] David Martin Powers. “Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation.” In: (2011).

[257] J. Ross Quinlan. “Induction of decision trees.” In: *Machine learning* 1.1 (1986), pp. 81–106.

[258] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[259] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Dave W. Binkley. "Recovering test-to-code traceability using slicing and textual analysis." In: *Journal of Systems and Software* 88 (2014), pp. 147–168. DOI: 10.1016/j.jss.2013.10.019. URL: https://doi.org/10.1016/j.jss.2013.10.019.

[260] Abdallah Qusef, Mahmoud O Elish, and David Binkley. "An Exploratory Study of the Relationship Between Software Test Smells and Fault-Proneness." In: *IEEE Access* 7 (2019), pp. 139526–139536.

[261] Yahya Rafique and Vojislav B Mišić. "The effects of test-driven development on external quality and productivity: A meta-analysis." In: *IEEE Transactions on Software Engineering* 39.6 (2013), pp. 835–856.

[262] Foyzur Rahman and Premkumar Devanbu. "How, and why, process metrics are better." In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 432–441.

[263] A Ananda Rao and K Narendar Reddy. "Detecting bad smells in object oriented design using design change propagation probability matrix 1." In: (2007).

[264] D Rapu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. "Using history information to improve design flaws detection." In: *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.* IEEE. 2004, pp. 223–232.

[265] Gema Rodríguez-Pérez, Gregorio Robles, and Jesús M González-Barahona. "Reproducibility and Credibility in Empirical Software Engineering: A Case Study based on a Systematic Literature Review of the use of the SZZ algorithm." In: *Information and Software Technology* 99 (2018), pp. 164–176.

[266] Peter H Rossi, James D Wright, and Andy B Anderson. *Handbook of survey research*. Academic Press, 2013.

[267] Pasquale Salza, Fabio Palomba, Dario Di Nucci, Andrea De Lucia, and Filomena Ferrucci. "Third-party libraries in mobile apps." In: *Empirical Software Engineering* (2019), pp. 1–37.

[268] A. J. Scott and M. Knott. “A cluster analysis method for grouping means in the analysis of variance.” In: *Biometrics* 30 (1974), pp. 507–512.

[269] Claude E Shannon. “Prediction and entropy of printed English.” In: *Bell system technical journal* 30.1 (1951), pp. 50–64.

[270] Samuel Sanford Shapiro and Martin B Wilk. “An analysis of variance test for normality (complete samples).” In: *Biometrika* 52.3/4 (1965), pp. 591–611.

[271] Martin Shepperd, David Bowes, and Tracy Hall. “Researcher bias: The use of machine learning in software defect prediction.” In: *IEEE Transactions on Software Engineering* 40.6 (2014), pp. 603–616.

[272] Emad Shihab, Zhen Ming Jiang, Walid M Ibrahim, Bram Adams, and Ahmed E Hassan. “Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project.” In: *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. 2010, pp. 1–10.

[273] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. “Why we refactor? confessions of github contributors.” In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2016, pp. 858–870.

[274] Davi Bernardo Silva, Andre Takeshi Endo, Marcelo Medeiros Eler, and Vinicius HS Durelli. “An analysis of automated tests for mobile Android applications.” In: *2016 XLII Latin American Computing Conference CLEI*. IEEE. 2016, pp. 1–9.

[275] Frank Simon, Frank Steinbruckner, and Claus Lewerentz. “Metrics based refactoring.” In: *Proceedings fifth european conference on software maintenance and reengineering*. IEEE. 2001, pp. 30–38.

[276] Chris Simons, Jeremy Singer, and David R White. “Search-based refactoring: Metrics are not enough.” In: *International Symposium on Search Based Software Engineering*. Springer. 2015, pp. 47–61.

[277] Dag IK Sjoberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dyba. "Quantifying the effect of code smells on maintenance effort." In: *IEEE Transactions on Software Engineering* 8 (2013), pp. 1144–1156.

[278] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. "When do changes induce fixes?" In: *ACM sigsoft software engineering notes*. Vol. 30. 4. ACM. 2005, pp. 1–5.

[279] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. "Pydriller: Python framework for mining software repositories." In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. 2018, pp. 908–911.

[280] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. "On the relation of test smells to software code quality." In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2018, pp. 1–12.

[281] Charles Spearman. "The proof and measurement of association between two things." In: *American journal of Psychology* 15.1 (1904), pp. 72–101.

[282] Diomidis Spinellis. "Tool writing: a forgotten art?(software tools)." In: *IEEE Software* 22.4 (2005), pp. 9–11.

[283] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L Bleris. "Code quality analysis in open source software development." In: *Information Systems Journal* 12.1 (2002), pp. 43–60.

[284] Statista. *Number of smartphone users worldwide*. Mar. 2020.

[285] Mark Steyvers and Tom Griffiths. "Probabilistic topic models." In: *Handbook of latent semantic analysis*. Psychology Press, 2007, pp. 439–460.

[286] Mervyn Stone. "Cross-validatory choice and assessment of statistical predictions." In: *Journal of the royal statistical society. Series B (Methodological)* (1974), pp. 111–147.

[287] Jaymie Strecker and Atif M Memon. "Accounting for defect characteristics in evaluations of testing techniques." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 21.3 (2012), p. 17.

[288] Davide Taibi, Andrea Janes, and Valentina Lenarduzzi. "How developers perceive smells in source code: A replicated study." In: *Information and Software Technology* 92 (2017), pp. 223–235.

[289] Damian A. Tamburri, Rick Kazman, and Hamed Fahimi. "The Architect's Role in Community Shepherding." In: *IEEE Software* 33.6 (2016), pp. 70–79.

[290] Damian A Tamburri, Philippe Kruchten, Patricia Lago, and Hans van Vliet. "What is social debt in software engineering?" In: *Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop on*. 2013, pp. 93–96. DOI: 10.1109/CHASE.2013.6614739.

[291] Damian A Tamburri, Fabio Palomba, and Rick Kazman. "Success and Failure in Software Engineering: A Followup Systematic Literature Review." In: *IEEE Transactions on Engineering Management* (2020).

[292] Damian Andrew Andrew Tamburri, Fabio Palomba, and Rick Kazman. "Exploring Community Smells in Open-Source: An Automated Approach." In: *IEEE Transactions on Software Engineering* (2019).

[293] Chakkrit Tantithamthavorn and Ahmed E Hassan. "An experience report on defect modelling in practice: Pitfalls and challenges." In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. 2018, pp. 286–295.

[294] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, Akinori Ihara, and Kenichi Matsumoto. "The impact of mislabelling on the performance and interpretation of defect prediction mod-

els." In: *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*. Vol. 1. IEEE. 2015, pp. 812–823.

[295] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. "An empirical comparison of model validation techniques for defect prediction models." In: *IEEE Transactions on Software Engineering* 43.1 (2017), pp. 1–18.

[296] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. "The impact of automated parameter optimization on defect prediction models." In: *IEEE Transactions on Software Engineering* 45.7 (2018), pp. 683–711.

[297] David Martinus Johannes Tax. "One-class classification: Concept learning in the absence of counter-examples." In: (2002).

[298] New York Times. *How COVID19 has changed social interactions*. Mar. 2020.

[299] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. "JDeodorant: Identification and removal of type-checking bad smells." In: *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*. IEEE. 2008, pp. 329–331.

[300] Nikolaos Tsantalis and Alexander Chatzigeorgiou. "Identification of move method refactoring opportunities." In: *IEEE Transactions on Software Engineering* 35.3 (2009), pp. 347–367.

[301] Nikolaos Tsantalis and Alexander Chatzigeorgiou. "Identification of extract method refactoring opportunities for the decomposition of methods." In: *Journal of Systems and Software* 84.10 (2011), pp. 1757–1782.

[302] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. "An empirical investigation into the nature of test smells." In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM. 2016, pp. 4–15.

[303] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. "There and back again: Can you compile that snapshot?" In: *Journal of Software: Evolution and Process* 29.4 (2017), e1838.

[304] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. "When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away)." In: *IEEE Transactions on Software Engineering* (2017).

[305] Bela Ujhazi, Rudolf Ferenc, Denys Poshyvanyk, and Tibor Gyimothy. "New conceptual coupling and cohesion metrics for object-oriented systems." In: *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. IEEE. 2010, pp. 33–42.

[306] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. "An empirical study of bugs in test code." In: *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE. 2015, pp. 101–110.

[307] Stef Van Buuren and Karin Groothuis-Oudshoorn. "mice: Multivariate imputation by chained equations in R." In: *Journal of statistical software* 45 (2011), pp. 1–67.

[308] Arie Van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. "Refactoring test code." In: *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. 2001, pp. 92–95.

[309] Eva Van Emden and Leon Moonen. "Java quality assurance by detecting code smells." In: *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE. 2002, pp. 97–106.

[310] Bart Van Rompaey and Serge Demeyer. "Establishing traceability links between unit test cases and units under test." In: *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE. 2009, pp. 209–218.

[311] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. “On the detection of test smells: A metrics-based approach for general fixture and eager test.” In: *IEEE Transactions on Software Engineering* 33.12 (2007), pp. 800–817.

[312] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall. “Context is king: The developer perspective on the usage of static analysis tools.” In: *26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2018).

[313] Carmine Vassallo, Giovanni Grano, Fabio Palomba, Harald C Gall, and Alberto Bacchelli. “A large-scale empirical exploration on refactoring activities in open source software projects.” In: *Science of Computer Programming* 180 (2019), pp. 1–15.

[314] Carmine Vassallo, Fabio Palomba, Alberto Bacchelli, and Harald C Gall. “Continuous code quality: are we (really) doing that?” In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM. 2018, pp. 790–795.

[315] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. “How developers engage with static analysis tools in different contexts.” In: *Empirical Software Engineering* 25.2 (2020), pp. 1419–1457.

[316] Stephane Vaucher, Foutse Khomh, Naouel Moha, and Yann-Gaël Guéhéneuc. “Tracking design smells: Lessons from a study of god classes.” In: *Reverse Engineering, 2009. WCRE’09. 16th Working Conference on*. IEEE. 2009, pp. 145–154.

[317] Santiago A Vidal, Claudia Marcos, and J Andrés Díaz-Pace. “An approach to prioritize code smells for refactoring.” In: *Automated Software Engineering* 23.3 (2016), pp. 501–532.

[318] Xiaoyin Wang, Yingnong Dang, Lu Zhang, Dongmei Zhang, Erica Lan, and Hong Mei. “Can I clone this piece of code here?” In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM. 2012, pp. 170–179.

[319] Tony Wasserman. “Software engineering issues for mobile application development.” In: (2010).

[320] Fadi Wedyan, Dalal Alrmuny, and James M Bieman. “The effectiveness of automated static analysis tools for fault detection and refactoring prediction.” In: *International Conference on Software Testing Verification and Validation*. 2009, pp. 141–150.

[321] Lili Wei, Yepang Liu, and Shing-Chi Cheung. “Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps.” In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2016, pp. 226–237.

[322] Yi Wei, Bertrand Meyer, and Manuel Oriol. “Is branch coverage a good measure of testing effectiveness?” In: *Empirical Software Engineering and Verification*. Springer, 2012, pp. 194–212.

[323] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. “Deep learning code fragments for code clone detection.” In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM. 2016, pp. 87–98.

[324] Frank Wilcoxon. “Individual comparisons by ranking methods.” In: *Biometrics bulletin* 1.6 (1945), pp. 80–83.

[325] Sue Wilkinson. “Focus group methodology: a review.” In: *International journal of social research methodology* 1.3 (1998), pp. 181–203.

[326] Claes Wohlin. “Guidelines for snowballing in systematic literature studies and a replication in software engineering.” In: *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. 2014, pp. 1–10.

[327] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[328] Zhou Xu, Jin Liu, Zijiang Yang, Gege An, and Xiangyang Jia. "The impact of feature selection on defect prediction performance: An empirical comparison." In: *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*. IEEE. 2016, pp. 309–320.

[329] Aiko Yamashita and Leon Moonen. "Do code smells reflect important maintainability aspects?" In: *2012 28th IEEE international conference on software maintenance (ICSM)*. IEEE. 2012, pp. 306–315.

[330] Aiko Yamashita and Leon Moonen. "Do developers care about code smells? an exploratory survey." In: *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE. 2013, pp. 242–251.

[331] Aiko Yamashita and Leon Moonen. "Exploring the impact of inter-smell relations on software maintainability: An empirical study." In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 682–691.

[332] Jiachen Yang, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. "Classification model for code clones based on machine learning." In: *Empirical Software Engineering* 20.4 (2015), pp. 1095–1125.

[333] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. "Better test cases for better automated program repair." In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017, pp. 831–841.

[334] Tao Ye and Shivkumar Kalyanaraman. "A recursive random search algorithm for large-scale network parameter configuration." In: *Proceedings of the 2003 ACM SIGMETRICS International conference on Measurement and modeling of computer systems*. 2003, pp. 196–205.

[335] Chak Shun Yu, Christoph Treude, and Mauricio Aniche. "Comprehending Test Code: An Empirical Study." In: (2019), to appear.

[336] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. "How open source projects use static code analysis tools in continuous integration pipelines." In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE. 2017, pp. 334–344.

[337] Nico Zazworka, Michele A Shaw, Forrest Shull, and Carolyn Seaman. "Investigating the impact of design debt on software quality." In: *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM. 2011, pp. 17–23.

[338] Jack Zhang, Shikhar Sagar, and Emad Shihab. "The evolution of mobile apps: An exploratory study." In: *Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile*. ACM. 2013, pp. 1–8.

[339] Min Zhang, Tracy Hall, and Nathan Baddoo. "Code bad smells: a review of current knowledge." In: *Journal of Software Maintenance and Evolution: research and practice* 23.3 (2011), pp. 179–202.

[340] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process." In: *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 2009, pp. 91–100.

[341] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. "Predicting defects for eclipse." In: *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. IEEE. 2007, pp. 9–9.