



UNIVERSIDADE
FEDERAL DE
ALAGOAS



UNIVERSIDADE FEDERAL DE ALAGOAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Dissertação de Mestrado

Understanding and Classifying Code Harmfulness

Rodrigo dos Santos Lima

rsl@ic.ufal.br

Orientador:

Baldoino Fonseca dos Santos Neto

MACEIÓ, FEVEREIRO DE 2020

Rodrigo dos Santos Lima

Understanding and Classifying Code Harmfulness

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Curso de Mestrado em Informática do Instituto de Computação da Universidade Federal de Alagoas.

Orientador:

Baldoino Fonseca dos Santos Neto

Maceió, Fevereiro de 2020

Catálogo na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico

Bibliotecária Responsável: Helena Cristina Pimentel do Vale – CRB4 - 661

L732u Lima, Rodrigo dos Santos.
Understanding and classifying code harmfulness / Rodrigo dos Santos
Lima. – 2020.
54 f. : il.

Orientador: Balduino Fonseca dos Santos Neto.
Dissertação (mestrado em Informática) - Universidade Federal de Alagoas.
Instituto de Computação. Maceió, 2020.

Bibliografia: f. 33-38.
Apêndices: f. 39-54.

1. Code smells. 2. Software – Qualidade. 3. Aprendizagem de máquina.
I. Título.

CDU: 004.4



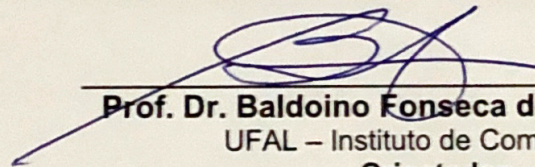
Folha de Aprovação

Rodrigo dos Santos Lima

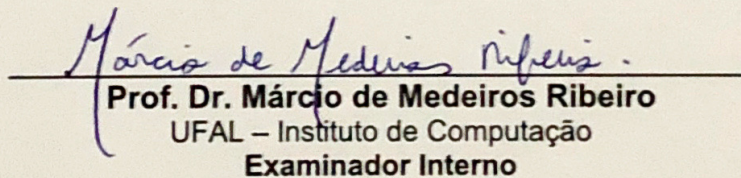
"Understanding and Classifying Code Harmfulness"

Dissertação submetida ao corpo docente do Programa de Pós-Graduação em Informática da Universidade Federal de Alagoas e aprovada em 28 de fevereiro de 2020.

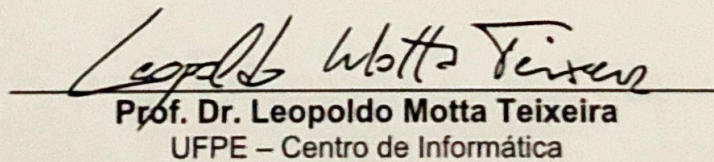
Banca Examinadora:



Prof. Dr. Balduino Fonseca dos Santos Neto
UFAL – Instituto de Computação
Orientador



Prof. Dr. Márcio de Medeiros Ribeiro
UFAL – Instituto de Computação
Examinador Interno



Prof. Dr. Leopoldo Motta Teixeira
UFPE – Centro de Informática

Agradecimentos

Nesses anos de mestrado, de muito estudo, esforço e empenho gostaria de agradecer a algumas pessoas que me acompanharam e foram fundamentais para realização de mais este sonho.

Aos meus pais Manoel e Neli por todo o apoio e incentivo ao longo dessa caminhada. Também por entender que a educação é a base fundamental do ser humano.

Agradeço ao meu orientador Prof. Dr. Balduino Fonseca dos Santos Neto por toda a paciência, apoio e críticas construtivas durante esse período. Seus ensinamentos foram fundamentais para a minha formação como pesquisador.

Aos membros da banca, os professores Dr. Márcio de Medeiros Ribeiro e ao Dr. Leopoldo Motta Teixeira pela disposição e interesse em contribuir para a melhoria desse estudo.

Ao grupo de pesquisa EASY e seus integrantes que se tornaram colegas de trabalho e pessoas que pretendo ter amizade pelo resto da vida. Sem dúvida, fizeram parte da minha formação. Obrigado pelas discussões, cafezinhos e convivência diária no laboratório.

Agradeço também à Fundação de Amparo à Pesquisa do Estado de Alagoas (FAPEAL) pelo apoio financeiro para realização da pesquisa. Por último, a todos aqueles que direto ou indiretamente fizeram parte da minha formação: O meu muito obrigado!

Resumo

Code Smells geralmente indicam más opções de implementação que podem prejudicar a qualidade do *software*. Portanto, eles precisam ser detectados com cuidado para evitar degradação do *software*. Nesse contexto, alguns estudos tentam entender o impacto dos *Code Smells* na qualidade do *software*, enquanto outros propõem regras ou técnicas baseadas em aprendizado de máquina para detectar *Code Smells*. No entanto, nenhum desses estudos / técnicas se concentram na análise de trechos de código que são realmente prejudiciais à qualidade do *software*. Nosso estudo tem como objetivo entender e classificar a nocividade do código. Analisamos a nocividade em termos de código CLEAN, SMELLY, BUGGY e HARMFUL. Por código nocivo, queremos dizer código que já prejudicou a qualidade do *software* e ainda está sujeito a danos. Realizamos nosso estudo com 22 tipos de *Smells*, 803 versões de 12 projetos de código aberto, 40.340 bugs e 132.219 *Code Smells*. Os resultados mostram que, embora tenhamos um número alto de *Code Smells*, apenas 0,07% desses *Smells* são prejudiciais. O *Abstract Call From Constructor* é o tipo de *Smell* mais relacionado ao código nocivo. Para validar empiricamente nossos resultados, também realizamos uma pesquisa com 77 desenvolvedores. A maioria deles (90,4%) considera *Code Smells* prejudiciais ao software e 84,6% desses desenvolvedores acreditam que as ferramentas de detecção de *Code Smells* são importantes. Mas, esses desenvolvedores não estão preocupados em selecionar ferramentas capazes de detectar *Code Smells*. Também avaliamos técnicas de aprendizado de máquina para classificar a nocividade do código: elas atingem a eficácia de pelo menos 97% para classificar o código nocivo. Enquanto *Random Forest* é eficaz na classificação de *Code Smells* e nocivos, o *Gaussian Naïve Bayes* é a técnica menos eficaz. Nossos resultados também sugerem que as métricas de software e desenvolvedores são importantes para classificar códigos nocivos.

Palavras-chave: Code Smells, Qualidade de Software, Aprendizagem de Máquina

Abstract

Code smells typically indicate poor implementation choices that may degrade software quality. Hence, they need to be carefully detected to avoid such degradation. In this context, some studies try to understand the impact of code smells on the software quality, while others propose rules or machine learning-based techniques to detect code smells. However, none of those studies/techniques focus on analyzing code snippets that are really harmful to software quality. Our study aims to understand and classify code harmfulness. We analyze harmfulness in terms of CLEAN, SMELLY, BUGGY, and HARMFUL code. By harmful code, we mean code that has already harmed software quality and is still prone to harm. We perform our study with 22 smell types, 803 versions of 12 open-source projects, 40,340 bugs and 132,219 code smells. The results show that even though we have a high number of code smells, only 0.07% of those smells are harmful. The *Abstract Function Call From Constructor* is the smell type more related to harmful code. To cross-validate our results, we also perform a survey with 77 developers. Most of them (90.4%) consider code smells harmful to the software, and 84.6% of those developers believe that code smells detection tools are important. But, those developers are not concerned about selecting tools that are able to detect harmful code. We also evaluate machine learning techniques to classify code harmfulness: they reach the effectiveness of at least 97% to classify harmful code. While the **Random Forest** is effective in classifying both smelly and harmful code, the **Gaussian Naive Bayes** is the less effective technique. Our results also suggest that both software and developers' metrics are important to classify harmful code.

Keywords: Code Smells, Software Quality, Machine Learning

List of Figures

Figure 1 – Harmful Code.	9
Figure 2 – Study Design.	10
Figure 3 – Dataset Structure.	10
Figure 4 – Study Design Machine Learning Effectiveness.	12
Figure 5 – Switch Statements Smell.	18
Figure 6 – Bug Fix in code smell snippets.	19
Figure 7 – Long Parameter List.	19
Figure 8 – Survey Results.	21
Figure 9 – Results for Harmful Code (f1-score).	23
Figure 10 – Results for Smelly Code (f1-score).	24
Figure 11 – Most Important metrics among code smells (SHAP values).	25

List of Tables

Table 1 – Analyzed Projects.	6
Table 2 – Selected Code Smells and Tools	7
Table 3 – Code Category	15
Table 4 – Harmful Smell Types.	16

Contents

List of Figures	iv
List of Tables	v
Contents	vi
1 Introduction	1
1.1 Context and Problem	1
1.2 Objectives and Methodological Aspects	2
1.3 Contributions	2
1.4 Thesis Structure	3
2 Study Design	4
2.1 Research Questions	4
2.2 Projects Selection	5
2.3 Code Smells	5
2.4 Metrics	7
2.5 Locating Bug Fixes	8
2.6 Locating the buggy code	8
2.7 Locating Harmful Code	9
2.8 Dataset Structure	9
2.9 Calculating Effectiveness	10
2.10 Machine Learning Techniques	13
2.11 Evaluating the Importance of Features	13
2.12 Data Analysis	13
3 Results and Discussions	15
3.1 RQ₁ . To what extent are smells harmful to the software?	15
3.1.1 WHY IS THIS SMELL TYPE HARMFUL?	17
3.2 RQ₂ . Do developers perceive and handler code smells as harmful?	20
3.3 RQ₃ . How effective are Machine Learning techniques to classify harmful code?	22
3.4 RQ₄ . Which metrics are most influential on classifying harmful code?	24

4	Implications	27
5	Related Work	28
6	Threats to Validity	31
7	Conclusions and Future Works	32
	Bibliography	33
	APPENDIX A Supplementary Material	38
1	Survey	38
2	Tables: How effective are Machine Learning techniques to detect harmful code?	46
3	Table: Which metrics are most influential on detecting harmful code?	50
4	Software Metrics (CK)	53
5	Developer Metrics	54

1 Introduction

In this chapter, we present a summary of the research, starting with the context and problem, connecting them with the objectives and contributions of this work.

1.1 Context and Problem

During software development, developers perform changes to implement new requirements, to improve the code or to fix bugs. While those changes contribute to the evolution of the software, they may introduce code smells, which represent symptoms of poor design and implementation choices (FOWLER, 1999). The growing incidence of code smells is also an indicator of design degradation (MACIA et al., 2012), change and fault-proneness (PALOMBA et al., 2018) as well as it may hinder comprehensibility (ABBES et al., 2011). Thus, code smell detection is an elementary technique to improve the software longevity.

Several approaches have been proposed to identify *code smells* in an automatic way. For instance, some studies (Marinescu, 2004; MARINESCU, 2001; LANZA; MARINESCU, 2010; MUNRO, 2005) propose techniques that rely on smell detection rules defined by developers or reused from other projects or tools. Other studies (Di Nucci et al., 2018; AMORIM et al., 2016; Arcelli Fontana et al., 2016; Maiga et al., 2012; KESSENTINI et al., 2011; KHOMH et al., 2009) indicate that Machine Learning techniques (ML-techniques) such as *Decision Trees* (AMORIM et al., 2016; Arcelli Fontana et al., 2016), *Support Vector Machines* (Maiga et al., 2012), *Genetic Programming* (KESSENTINI et al., 2011), and *Bayesian Belief Networks* (KHOMH et al., 2009), are a promising way to automate part of the detection process, without asking the developers to define their own strategies of code smell detection. Even though studies (MACIA et al., 2012; PALOMBA et al., 2018; ABBES et al., 2011) have analyzed the impact of code smells and there are techniques to detect those smells (Marinescu, 2004; MARINESCU, 2001; LANZA; MARINESCU, 2010; MUNRO, 2005; Di Nucci et al., 2018; AMORIM et al., 2016; Arcelli Fontana et al., 2016; Maiga et al., 2012; KESSENTINI et al., 2011; KHOMH et al., 2009), none of those works focus on analyzing code snippets that are really harmful to software quality.

1.2 Objectives and Methodological Aspects

As explained above, to understanding and classifying code harmfulness is essential to identify pieces of code that were really harmful to the software, helping developers prioritize them while refactoring the code. Based on that, we set the following objectives.

This study is to understand and classify code harmfulness. We consider faults (in terms of bugs) as the main harmfulness aspect analyzed, since they are closely related to software failures. This way, we assess the harmfulness of code snippets in terms of four categories: CLEAN: there is no smell or bug historically associated with the code; SMELLY: code contains smells, but has no bug associated with it; BUGGY: code already had bugs associated with it, but no smell has been detected into the code; HARMFUL: code contains smells and it already had bugs associated with it. First, we analyze the occurrence of each category in open-source projects and which smell types are more related to harmful code. We also investigate the developers' perceptions regarding the harmfulness of code smells. Moreover, we evaluate the effectiveness of machine learning techniques to classify the harmfulness of code snippets, focusing on smelly and harmful code. Finally, we investigate which metrics are most influential in harmful code detection.

To perform the objectives above mentioned, we define an experiment with 22 smell types, 803 versions of 12 open-source projects, 40,340 bugs mined from GitHub issues and 132,219 code smells collected by the tools *Designite*, *PMD*, and *IntelliJ*. Along with the execution of the study we perform a survey with 77 developers.

1.3 Contributions

The main contributions of this study are:

- Identify which types of smells are more related to *Harmful Code*: Only a few types of smells are associated with bugs, *Abstract Function Call From Constructor* presents the highest association.
- Identify Developers' Perception about how harmful are code smells, where the vast majority of the developers (90.4%) consider code smells harmful to the software.

- Evaluate the effectiveness of ML-techniques to detect Harmful code in open-source projects, checking that ML-techniques classify harmful code as effective as they identify code smells.
- Identify which metrics are most influential on classifying harmful code, where the software metrics reached the highest importance. *Weight Method Class (WMC)* is the metric that presented the highest importance in most of the smells.
- A large dataset containing software metrics, developers metrics, smells, and bugs. Is available on our website (HARMFUL. . . , 2020).

From this study, we provide valuable information for the developers. The developers could focus their efforts on refactor only code smells that are really harmful to the software, reducing their efforts and the probability of a bug introducing change.

1.4 Thesis Structure

The rest of this thesis is structured as follows:

- Chapter 2 describes the design of our mixed study going over the research questions, how the data will be collected and analyzed.
- Chapter 3 presents the results and discussions of this study. We start analyzing the occurrence of CLEAN, SMELLY, BUGGY, and HARMFUL code in open-source projects as well as which smell types are more related to harmful code. Then, we investigate the developers' perceptions regarding the harmfulness of code smells. Then evaluate the effectiveness of machine learning techniques to detect harmful and smelly code and investigate which metrics are most important.
- Chapter 5 discuss the related works that have been done in the fields of the **Impact, Detection Tools and Techniques, Developers' Perceptions**. This chapter focuses more on comparing the existing works, contributions, and limitations.
- Chapter 6 details the limitations and threats to validity of this study.
- Chapter 7 presents our conclusion and ideas which could lead to future contributions in the field of Software Engineering.

2 Study Design

In this chapter, we describe the process that leads to the results of our study. First, we present the research questions of our work, and we present the steps of the study.

2.1 Research Questions

Several studies have investigated the impact of code smells (MACIA et al., 2012; PALOMBA et al., 2018; ABBES et al., 2011) as well as tools and techniques to detect smells (Marinescu, 2004; MARINESCU, 2001; LANZA; MARINESCU, 2010; MUNRO, 2005; Di Nucci et al., 2018; AMORIM et al., 2016; Arcelli Fontana et al., 2016; Maiga et al., 2012; KESSENTINI et al., 2011; KHOMH et al., 2009). However, none of those works focus on analyzing code snippets that are really harmful to software quality. In this context, we try to answer four main research questions:

RQ₁. *To what extent are smells harmful to the software?*

This research question assesses the presence of CLEAN, SMELLY, BUGGY, and HARMFUL code. As a result, we expect to understand the frequency of each category in open-source projects. This way, we can analyze whether smells might indicate bugs in such projects, as well as finding out the frequency of smells that are considered harmful. A high frequency of HARMFUL code indicates a close relation between smells and bugs. On the other hand, a low frequency of harmful code suggests that smells may not be a good proxy for bugs and, consequently, developers should focus their efforts on refactoring harmful code, instead of refactoring a high number of smells. After assessing HARMFUL code, we investigate which smell types are more harmful. The recognition of such smell types may help developers to be more cautious during software development to avoid these smell types.

RQ₂. *Do developers perceive and handle code smells as harmful?*

In this research question, we analyze the developers' perceptions about how harmful are code smells. This analysis may help us to understand if developers believe that code smells are really harmful to the software quality. Also, we investigate if developers handle code smells as harmful or not. In particular, we analyze if developers concern about using appropriate tools not only to detect smelly but also harmful code.

RQ₃. *How effective are Machine Learning techniques to detect harmful code?*

After analyzing the existence of harmful code in open-source projects and how harmful are code smells for developers, we evaluate the effectiveness of ML-techniques to detect HARMFUL code in open-source projects. Several studies have indicated ML-techniques as an effective way to detect smells (AMORIM et al., 2016; Arcelli Fontana et al., 2016; Arcelli Fontana; ZANONI, 2017), but there is no knowledge if they are as effective in detecting HARMFUL code. As a result, we expect to shed light toward the use of ML-techniques to detect smells that are really harmful to the software. This way, developers could use existing tools and techniques not only to detect smelly but also harmful code.

RQ₄. *Which metrics are most influential on detecting harmful code?*

Once we recognize the more effective Machine Learning techniques, we also analyze the metrics that have more influence to identify HARMFUL code. Identifying these influential metrics may help developers to avoid common code structures that are more likely to produce harmful code.

2.2 Projects Selection

We manually selected 12 Java projects according to the following criteria: (i) they must be open-source and hosted at GitHub; (ii) they must use the GitHub issues bug-tracking tool, to standardize our process of collecting bug reports (Section 2.5); and (iii) they must have had at least 3,000 smells and 20 bugs along their development.

Table 1 summarizes the characteristics of the selected projects. For each project, we analyze its complete history, from the first commit until the last one at the moment we collect the project information. The number of code smells ranges from 1,523 (*Apollo*) up to 121,165 (*Dubbo*), and the number of bugs varies from 22 (*Apollo*) up to 10,085 (*Okhttp*). The high numbers are due to analyzing the whole history of the projects.

2.3 Code Smells

In our study, we analyze 22 smell types, as reported in Table 2. We select these smell types because: (i) they affect different scopes, i. e., methods, and classes; (ii) they are also investigated in previous works on code smell detection (SHARMA; MISHRA; TIWARI, 2016; ALENEZI; ZAROOUR, 2018; FONTANA et al., 2015; KHOMH et al., 2009; KHOMH et al.,

Table 1 – Analyzed Projects.

Project Name	Domain	# Versions	# Smells	# Bugs
Acra	Application	52	3,470	163
ActiveAndroid	Library	1	141	754
Aeron	Application	76	10,914	1,634
Aerosolve	Library	100	1,863	443
Apollo	Library	32	1,524	27
Butterknife	Library	45	1,636	861
Dubbo	Framework	62	27,843	2,216
Ethereumj	Cryptocurrency	50	9,169	2,763
Ganttproject	Application	31	8,275	4,365
Joda-time	Library	64	7,421	684
Lottie-android	Library	75	4,143	58
Okhttp	Library	80	9,412	10,111
Spring-boot	Framework	136	46,551	40

2011; Maiga et al., 2012; AMORIM et al., 2016; Arcelli Fontana et al., 2016); (iii) they have detection rules defined in the literature or implemented in available tools (HARMFUL. . . , 2020; DESIGNITE. . . , 2020; PMD. . . , 2020); and (iv) studies (PALOMBA et al., 2018; MA et al., 2018; HALL et al., 2014) indicate a negative impact of those smell types on software quality.

To collect code smells, we use three smell detection tools: *Designite*, *PMD*, and *IntelliJ*. These tools have already been used in previous studies (SHARMA; MISHRA; TIWARI, 2016; ALENEZI; ZAROOUR, 2018; FONTANA et al., 2015). We execute these tools for each version of the software, in the entire history of the repository. For each class and method, we register the smells associated with it along with the software versions. One might argue that smell may be registered several times to the same class or method along with the versions. To mitigate this issue, we only count the smell if a class or method has changed when compared to the previous version. Furthermore, we conduct a careful manual validation on a sample of smells detected. Two pairs of researchers (familiar with code smells detection) from our research lab validated this sample. Each pair was responsible for a fraction of the sample, and each individual validated the same candidate smells. We conducted the validation due to the high numbers of false-positives (changes reported as bug-introducing when they are actually not) reported in previous studies (KIM et al., 2006; WILLIAMS; SPACCO, 2008; GRIFFOR et al., 2017). In Table 2, we present names, scope (i.e., method or class), and the tool used to collect each smell. More details about the instrumentation used by each tool to detect the analyzed smell types are available on our website (HARMFUL. . . , 2020).

Table 2 – Selected Code Smells and Tools

Code Smell	Affected Code	Tool
Broken Hierarchy Cyclic-Dependent Modularization Deficient Encapsulation Imperative Abstraction Insufficient Modularization Multifaceted Abstraction Rebellious Hierarchy Unnecessary Abstraction Unutilized Abstraction	Class	Designite
Long Method Long Parameter List Long Statement Switch Statements Simplified Ternary Abst Func Call From Const Magic Number Missing default Missing Hierarchy Unexploited Encapsulation Empty catch clause	Method	PMD / Designite / IntelliJ Designite
Long Identifier	Class / Method	PMD / Designite / IntelliJ

2.4 Metrics

For each version of the analyzed projects, we compute metrics related to the source code and developers involved.

Source Code Metrics. We collect 52 metrics at class and method level, covering six quality aspects of object-oriented software: complexity, cohesion, size, coupling, encapsulation, and inheritance. We chose the CK (Chidamber and Kemerer) metrics (CHIDAMBER; KEMERER, 1994), which are well-known and have been used by previous studies to detect code smells (KHOMH et al., 2011; YAMASHITA; MOONEN, 2013; PALOMBA et al., 2013; PALOMBA et al., 2015; HOZANO et al., 2017; AMORIM et al., 2016). More details about the metrics, as well as the tools used to collect them, are described at our website (HARMFUL. . . , 2020) and appendix A.

Developer Metrics. Open-source environments, such as GitHub, made it easier for developers with different technical capabilities and social interactions to actively and simultaneously contribute to the same project. In such environments, developers can perform a variety of activities, such as committing code, opening and closing pull requests and issues, and discussing contributions. Even though developers can collaborate on different projects, their technical capabilities and social interactions can be determinant factors to the software

quality. For example, a novice developer can introduce bugs when performing some change. Developer metrics may be a promising way to recognize which developers' actions may lead to bugs. To perform our study, we collect 14 previously described developer metrics (FALCÃO et al., 2018), related to three main factors: experience, contribution, and social aspects (i.e., number of followers). We collect metrics using *PyDriller* (SPADINI; ANICHE; BACCHELLI, 2018), a tool to support Git repository analysis. More details about the developer metrics are described at our website (HARMFUL... , 2020) and appendix A.

2.5 Locating Bug Fixes

GitHub issues are useful to keep track of tasks, enhancements, and bugs related to a project. Furthermore, developers can label issues to characterize them. For example, an issue opened to fix a bug would typically be associated with the bug label. After fixing the bug, the issue can be closed. To collect the reports of fixed bugs in the selected projects, we mined the closed issues related to bugs in each project. To identify these issues, we verified whether they contained the bug or defect labels. As a result of this process, we collected 40,340 bug reports from the 12 analyzed projects. A similar approach was made in recent studies (SANTOS et al., 2020; KHOMH et al., 2011; PALOMBA et al., 2018).

2.6 Locating the buggy code

GitHub provides the functionality to close an issue using commit messages. For example, prefacing a commit message with keywords “Fixes”, “Fixed”, “Fix”, “Closes”, “Closed” or “Close”, followed by an issue number, such as, “Fixes #12345”, will automatically close the issue when the commit is merged into the master branch. This way, when this strategy is used to close a bug issue, we assume the commit that closed the issue as being the bug fixing commit.

Associating a bug (issue) with a commit fixing allows us to identify the methods and classes that were modified to fix the bug. This way, we conservatively establish that in the immediate previous commit to the fix, these methods and classes are considered *buggy*. We perform a similar approach by previous studies (PALOMBA et al., 2018; OLBRICH; CRUZES; SJOØBERG, 2010; KHOMH et al., 2012; SANTOS et al., 2020), which consists

of assuming that the class or method is directly or indirectly linked to the bug if they were modified in the commit fix.

2.7 Locating Harmful Code

We consider a code snippet that already had bugs associated with it and still contains smells as HARMFUL. To identify harmful code, we first collect the code smells detected along with the software history (from the first software version until the current one). Then, we verify which code smells already had bugs associated to it. As a result, we obtain the code snippets that contain smells and already had bugs associated to it, i.e., the harmful code snippets represented in Figure 2.

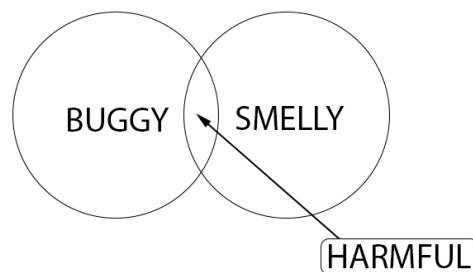


Figure 1 – Harmful Code.

The Figure 2 presents the steps made to build the dataset.

2.8 Dataset Structure

After collecting metrics and classifying code snippets, we build our dataset, structured as Figure 3 presents. Each instance in the dataset represents a code snippet (*class* or *method*) extracted from the analyzed projects. Associated with each code snippet, we have the software and developer metrics as well as its harmfulness (CLEAN, SMELLY, BUGGY, or HARMFUL). While the metrics represent characteristics (e. g., number of lines of code), the harmfulness (CLEAN, SMELLY, BUGGY and HARMFUL) indicates the category that the code snippet belongs. Our dataset is composed of 28,371,822 instances of code snippets, containing 28,216,238 CLEAN, 132,219 SMELLY, 40,340 BUGGY and 1,048 HARMFUL.

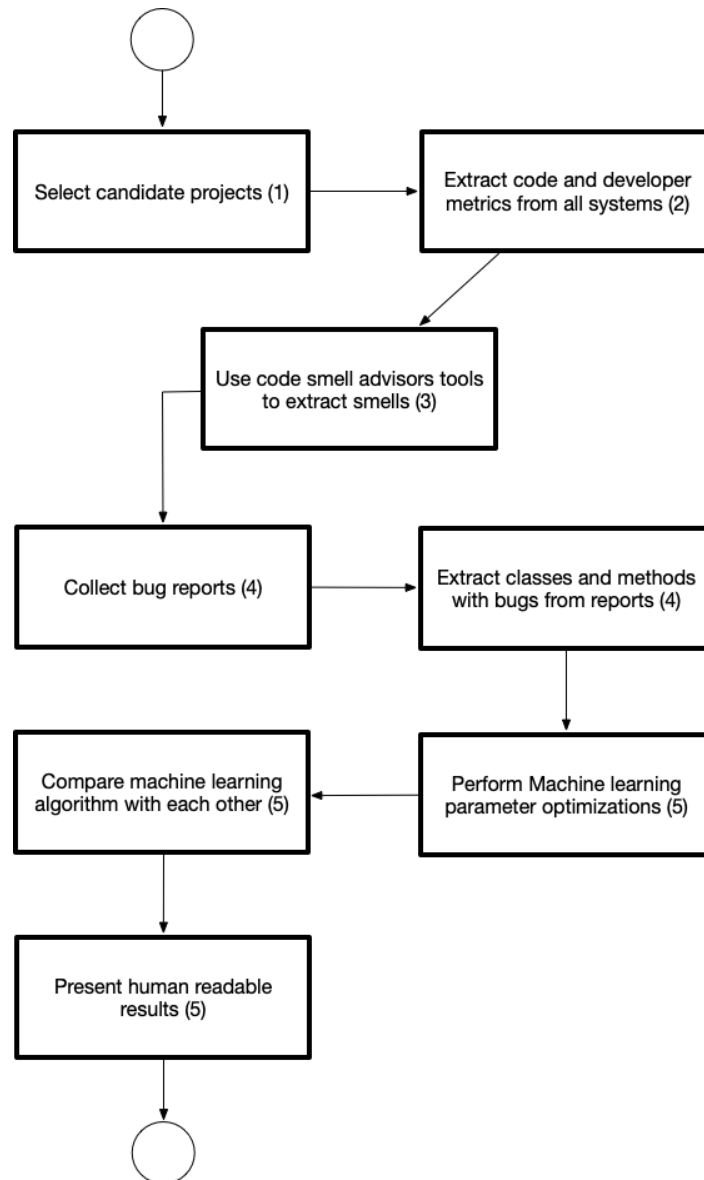


Figure 2 – Study Design.

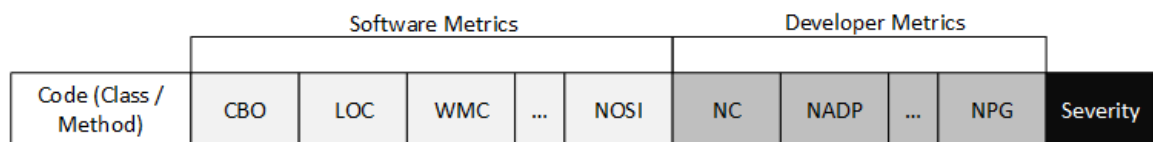


Figure 3 – Dataset Structure.

2.9 Calculating Effectiveness

We evaluate the effectiveness of seven ML-techniques to classify smelly and harmful code as follows. First, we split the dataset into 11 smaller datasets, one for each smell. They

are extremely imbalanced, i.e., the percentage of CLEAN code is higher compared to the other categories (SMELLY, HARMFUL, BUGGY). Class imbalance can result in a serious bias towards the majority class, thus reducing the evaluation or classifications tasks (GUEHENEUC; SAHRAOUI; ZAIDI, 2004). To avoid this, we apply an under-sampling technique that consists on randomly and uniformly under-sampling the majority across other classes, similar to a recent study (Arcelli Fontana; ZANONI, 2017).

We chose under-sampling instead of over-sampling to avoid the generation of artificial instances of SMELLY, HARMFUL, and BUGGY, as performed by over-sampling techniques.

Next, we preprocess the dataset to avoid inconsistencies and to remove irrelevant and redundant features (Arcelli Fontana et al., 2016; Arcelli Fontana; ZANONI, 2017; AMORIM et al., 2016). In particular, we normalize metrics with the default range between 0 and 1. Moreover, if two metrics present a Pearson correlation higher than 0.8, we randomly remove one of them.

After performing the preprocessing, we split each dataset into two sets: training and testing set containing $\frac{2}{3}$ and $\frac{1}{3}$ of the entire data respectively. We use the training data to learn the hyper-parameter configurations of the ML-techniques by applying a stratified 5-fold cross-validation procedure to ensure that each fold has the same proportion of observations with a given class outcome value, as previous studies do (Arcelli Fontana; ZANONI, 2017; Arcelli Fontana et al., 2016). Searching for the best hyper-parameter configuration for each technique is a complex, time-consuming, and challenging task. Each technique has a set of hyper-parameters that can be of different types, i.e., continuous, ordinal, and categorical, making it more difficult to calibrate them, due to the large space of hyper-parameters. The Grid Search algorithm (BERGSTRA; BENGIO, 2012) is the most common way to explore different configurations for each ML-technique. However, it would be very time consuming to explore the entire configuration search space for each technique. To avoid this, we tune the hyper-parameters using Hyperopt-Sklearn (KOMER; BERGSTRA; ELIASMITH, 2014),¹ which consists of an automatic hyper-parameter configuration for the Scikit-Learn Machine Learning library. Hyperopt-Sklearn uses Hyperopt (BERGSTRA; YAMINS; COX, 2013) to describe a search space over possible parameters configurations for different ML-techniques using Bayesian optimization (KOMER; BERGSTRA; ELIASMITH, 2014). The

¹ <<https://github.com/hyperopt/hyperopt-sklearn>>

use of Bayesian optimization algorithms is very effective and less time-consuming than the traditional approaches (KOMER; BERGSTRA; ELIASMITH, 2014).

Once we learn the hyper-parameters configuration of the techniques, we evaluate the effectiveness of these techniques on unseen data — testing data. Previous studies (AMORIM et al., 2016) have used this procedure to avoid overfitting of the techniques, i. e., to avoid that, a technique becomes too specific to the training data, without it being generalized to unseen data. We measure effectiveness in terms of f-measure, which is typically used for evaluating the performance of classification tasks, providing a more realistic performance measure of a test, since it is computed by the harmonic mean of the precision and recall, the Figure 4 presents these steps.

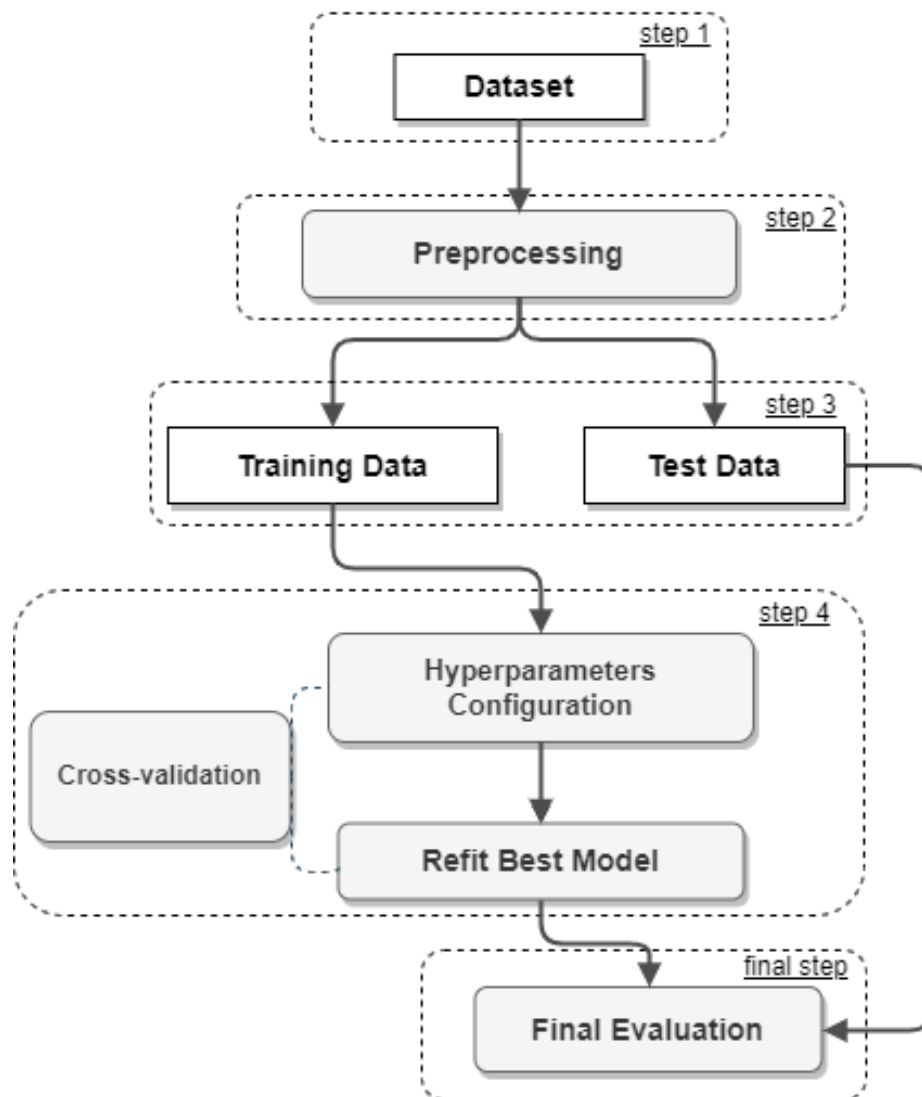


Figure 4 – Study Design Machine Learning Effectiveness.

2.10 Machine Learning Techniques

In this study, we apply a variety of techniques based on popular algorithms that had a good performance in previous studies (AZEEM et al., 2019) on classifying code smells. To perform that, we use the scikit-learn (PEDREGOSA et al., 2011) python library in the following ML-techniques: K-nearest Neighbors (KNN) (ALTMAN, 1992); Decision Tree (BREIMAN, 1984); Random Forest (HO, 1995); Gradient Boosting (BREIMAN et al., 1984); Gaussian Naive Bayes (HAND; YU, 2001); Support Vector Machines (SVM) (BOSER; GUYON; VAPNIK, 1992) and Ada Boost (SCHAPIRE, 1999)

2.11 Evaluating the Importance of Features

To evaluate the importance of each metric (feature) in the effectiveness of the ML-techniques analyzed, we use the *Shapley Additive Explanations* (SHAP) framework. It consists of a unified approach to explain the output of any ML model (LUNDBERG; LEE, 2017). The SHAP framework uses game theory with local explanations to give an importance value to each feature used by the ML-technique that had the highest score to detect HARMFUL code. This way, we can find out which metrics are more important in the detection of HARMFUL code category.

2.12 Data Analysis

To answer **RQ₁**, we analyze the frequency of CLEAN, SMELLY, BUGGY, and HARMFUL code in the projects. Moreover, we assess the proportion of HARMFUL code when compared to the number of smells. Such analysis helps us to understand how close is the relationship is between SMELLY and BUGGY code. We further investigate the HARMFUL code by analyzing which smell types are more harmful.

In **RQ₂**, we use an online survey to collect developers' perceptions on the harmfulness of code smells. Also, we investigate if developers concerns on selecting tools able to detect not only smelly but also harmful code.

RQ₃ analyzes the effectiveness of ML-techniques to recognize smelly and harmful code. In both cases, we apply the under-sampling method to balance the number of instances between the positives classes (i. e., smelly and harmful code) and negative classes associated

with CLEAN and BUGGY code. A similar method has been used in previous studies (Arcelli Fontana; ZANONI, 2017) to obtain a balanced dataset. Next, we use the balanced dataset to evaluate the ML-techniques through a 5-fold cross-validation procedure. Finally, we evaluate the effectiveness of each technique by using the *f1-score* metric in unseen data.

Finally, to answer **RQ₄**, we analyze which metrics are more important to the prediction model produced from the most effective ML-technique to detect HARMFUL code. Such importance is calculated using SHAP values (LUNDBERG; LEE, 2017), which is a measure to evaluate the impact of each feature on the output of the ML-technique. From this analysis, we can identify code implementations that are more related to HARMFUL code and, consequently, help developers to avoid those implementations.

3 Results and Discussions

In this chapter, we describe and discuss the main results of the study. We structure the data presentation and discussion in terms of our four research questions.

3.1 RQ₁. To what extent are smells harmful to the software?

In this research question, we analyze the code smells harmfulness by assessing the proportion of smells that are associated with bugs. Table 3 presents the number and proportion of code snippets considered as CLEAN, SMELLY, BUGGY, or HARMFUL in the analyzed projects.

CLEAN. All the analyzed projects present more than 96% of CLEAN code. This high proportion is expected since we analyze all software versions along with its evolution history. Although we have been careful by avoiding unchanged code snippets between versions, the code snippets are frequently changed over time. This high proportion has pros and cons. **High proportion means that developers could focus their efforts on a small part of the source code to identify smells or bugs.** On the other hand, a lower proportion could mean that the software has many smells or bugs, which can be harmful to quality but can also be useful for researchers to better understand inherent characteristics of this type of code.

Table 3 – Code Category

Project	# Clean	# Smelly	# Buggy	# Harmful
Acra	195,874 (98.18%)	3,470 (1.74%)	141 (0.07%)	22 (0.01%)
Aeron	1,861,703 (99.33%)	10,914 (0.58%)	1,489 (0.08%)	145 (0.01%)
Aerosolve	532,715 (99.57%)	1,863 (0.35%)	308 (0.06%)	135 (0.03%)
Apollo	313,379 (99.51%)	1,523 (0.48%)	22 (0.01%)	5 (0%)
Butterknife	74,153 (96.74%)	1,636 (2.13%)	857 (1.12%)	4 (0.01%)
Dubbo	2,854,863 (98.96%)	27,843 (0.97%)	1,939 (0.07%)	277 (0.01%)
Ethereumj	1,690,701 (99.3%)	9,168 (0.54%)	2,638 (0.15%)	125 (0.01%)
Ganttproject	4,263,646 (99.7%)	8,275 (0.19%)	4,077 (0.1%)	288 (0.01%)
Joda-time	3,021,538 (99.73%)	7,420 (0.24%)	676 (0.02%)	8 (0%)
Lottie-android	499,291 (99.17%)	4,143 (0.82%)	52 (0.01%)	6 (0%)
Okhttp	1,422,553 (98.65%)	9,412 (0.65%)	10,085 (0.7%)	26 (0%)
Spring-boot	11,485,822 (99.6%)	46,551 (0.4%)	33 (0%)	7 (0%)

SMELLY. Different from CLEAN code, the projects present a low proportion of SMELLY code. The projects have at most 2.13%. Although the proportion of SMELLY code is lower than CLEAN, smells are still present in a large portion of the code snippets. The number of smelly code snippets in the projects varies from 1,523 (*Apollo*) up to more than 27 thou-

sand (*Dubbo*). Such results indicate that **code smells detection tools still identify a large number of smells, making it difficult for developers to use them in practice.**

BUGGY. The proportion of BUGGY code is lower than SMELLY code. The projects contains a proportion lower than 2%. But, this low proportion does not mean a low number of BUGGY code snippets. The projects contain from 22 (*Apollo*) up to more than 10,085 (*Okhttp*) BUGGY code snippets. Which means that **developers still have to concern with a considerable number of BUGGY code snippets to inspect.**

HARMFUL. Similarly to BUGGY code, the projects also present low proportion of HARMFUL code. The number of HARMFUL code snippets is much lower than the number of BUGGY ones, varying from only four (*Butterknife*) up to 288 (*Ganttproject*). This suggests that **even though existing tools detect a large number of smells, there is still a high number of bugs that are not related to those smells.** Such results reinforce a previous study (Palomba et al., 2014), indicating that only a few smells types are considered as harmful by developers. Hence, we also analyze the smell types more related to SMELLY and HARMFUL code. Table 4 describes the smell types analyzed in our study, the number of smells, and harmful code associated with each type.

Table 4 – Harmful Smell Types.

Smell Type	# Code Smells	# Harmful Code
Abstract Function Call From Constructor	118 (92.2%)	10 (7.8%)
Broken Hierarchy	18,152 (99.8%)	30 (0.2%)
Cyclic-Dependent Modularization	20,448 (99.6%)	79 (0.4%)
Deep Hierarchy	88 (98.9%)	1 (1.1%)
Deficient Encapsulation	31,774 (99.8%)	56 (0.2%)
Empty catch clause	8,116 (99.7%)	23 (0.3%)
Imperative Abstraction	1,100 (98%)	23 (2%)
Insufficient Modularization	20,351 (99.6%)	75 (0.4%)
Long Identifier	4,588 (99.7%)	15 (0.3%)
Long Method	11,371 (99.4%)	70 (0.6%)
Long Parameter List	10,839 (99.4%)	61 (0.6%)
Long Statement	52,701 (99.7%)	169 (0.3%)
Magic Number	40,388 (99.6%)	155 (0.4%)
Missing default	3,888 (99.7%)	12 (0.3%)
Missing Hierarchy	908 (99.8%)	2 (0.2%)
Multifaceted Abstraction	885 (99.4%)	5 (0.6%)
Rebellious Hierarchy	671 (99.3%)	5 (0.7%)
Simplified Ternary	314 (98.4%)	5 (1.6%)
Switch Statements	7,740 (99.3%)	58 (0.7%)
Unexploited Encapsulation	688 (99.7%)	2 (0.3%)
Unnecessary Abstraction	665 (99.7%)	2 (0.3%)
Unutilized Abstraction	128,751 (99.9%)	188 (0.1%)

Smell Types Harmfulness. *Abstract Function Call From Constructor* presents the highest percentage of HARMFUL code, reaching 7.8% of HARMFUL smells. Both *Imper-*

ative Abstraction and *SimplifiedTernary* present a slightly lower percentage than *Abstract Function Call From Constructor*, reaching 2% and 1.6% of harmful code, respectively. While the *Switch Statements*, *Rebellious Hierarchy*, *Long Method*, *Long Parameter List* and *Switch Statements* present percentage between 0.6% and 0.7%, the *Magic Number*, *Cyclic-dependent Modularization* and *Long Identifier* present percentage between 0.3% and 0.4%. Even though some smell types present a percentage higher 1%, in half of the smell types analyzed the percentage of harmful code is close to zero.

3.1.1 WHY IS THIS SMELL TYPE HARMFUL?

We observe that some Smell Types are more harmful than others, they may lead to bug proneness.

Long Statement. This smell occurs when a statement is excessively lengthy, it is a sign that your branches in the switch are doing too much, this can be the main reason for the large number of harmful code (SURYANARAYANA; SAMARTHYAM; SHARMA, 2015d).

Abstract Call From Constructor. Constructors should only call non-overridable methods, calling an overridable method from a constructor could result in failures or strange behaviour when instantiating a subclass which overrides the method (SURYANARAYANA; SAMARTHYAM; SHARMA, 2015d). This is why the abstract call from constructor present the highest percentage of harmful code.

Switch Statements. Case statement is used for conditional operations. Sometimes, it is considered in category of code smells. In some cases switch statements provides redundant code, similar switch statements are scattered throughout a program (FOWLER, 1999). If you add or remove a clause in one switch, you often have to find and repair the others too. In this way most of the harmful occurs when the developer forget to repair some of the copies of the switch. The Figure 5 shows a change to fix a bug (#1255) of the project GanttProject, the bug was on a piece of the code that has a Switch Statement smell and the developer repaired it in all the copies of the code.

Long Method. It happens when a method contains too many lines of code, it is almost always a violation of single responsibility principle (FOWLER, 1999). It's is always hard to test as the consequence of the fact that we coupling too many objects. This could be the

```

248 239 +      switch (dayMask & GPCalendar.DayMask.WORKING) {
249 240      case 0:
250 241          if (event == null) {
251 -      result.add(CalendarEventAction.addException(getProject().getActiveCalendar(), date));
242 +      result.add(CalendarEventAction.addException(getProject().getActiveCalendar(), date, getUndoManager()));
252 243          }
253 244          break;
254 245          case GPCalendar.DayMask.WORKING:
255 -      result.add(CalendarEventAction.removeException(getProject().getActiveCalendar(), date));
246 +      result.add(CalendarEventAction.removeException(getProject().getActiveCalendar(), date, getUndoManager()));
256 247          break;
257 248          }
258 249          }
259 250          if ((dayMask & GPCalendar.DayMask.HOLIDAY) != 0) {
260 -      result.add(CalendarEventAction.removeHoliday(getProject().getActiveCalendar(), date));
251 +      result.add(CalendarEventAction.removeHoliday(getProject().getActiveCalendar(), date, getUndoManager()));
261 252          } else {
262 253          if (event == null) {
263 -      result.add(CalendarEventAction.addHoliday(getProject().getActiveCalendar(), date));
254 +      result.add(CalendarEventAction.addHoliday(getProject().getActiveCalendar(), date, getUndoManager()));
264 255          }
265 256          }
266 257          }

```

Figure 5 – Switch Statements Smell.

main reason of the harmful presence, as the long method is not well tested in most cases it leads to the propagation of bugs. The Figure 6 show a bug fix in two classes, in the first `CompatibleTypeUtils` in the method `compatibleTypeConvert` where the code was repaired we have many code smells: the method itself is a *Long Method*, it has many *Switch Statements* in the whole method, there is *Magic Numbers* on lines 57, 61, 109.

Long Parameter List. Long Parameter List indicates that there might be something wrong with the implementation, it is hard to use a method call or to get the parameters in the correct order, usually, this method calls has null parameters as optional parameters (FOWLER, 1999). Usually, it is no clear what each parameter does. The bugs in these methods are caused by changing the order of the parameters or not sending it correctly, that's why a long parameter list has a large number of harmful. Figure 7 shows a method `recalculateActivities` from the project `GanttProject` that contains this smell, it is called in many places of the application for example in the class `TaskImpl.java` on lines (200, 681, 994 and 1022) each call passing different parameters in different order, this method was directly related to a bug of the issue: **#1413** fixed by the commit **156c4dc**.

Deficient Encapsulation. This smell occurs when the declared accessibility of one or more members of abstraction is more permissive than actually required. For example, a class that makes its field public suffers from Deficient Encapsulation (SURYANARAYANA;

fix issue 4328, fix PojoUtils realize issue #4334 #4521

Merged beiwei30 merged 3 commits into apache:master from nullcodeexecutor:master on 16 Jul 2019

Conversation 1 Commits 3 Checks 0 Files changed 3 +869 -852

Changes from all commits File filter... Jump to... 0 / 3 files viewed Review changes

2 dubbo-common/src/main/java/org/apache/dubbo/common/utils/CompatibleTypeUtils.java

```

@@ -129,6 +129,8 @@ public static Object compatibleTypeConvert(Object value, Class<?> type) {
    129     129         return BigDecimal.valueOf(number.doubleValue());
    130     130     } else if (type == Date.class) {
    131     131         return new Date(number.longValue());
    132     132     } else if (type == boolean.class || type == Boolean.class) {
    133     133     +         return 0 != number.intValue();
    134     134     }
    135     135     } else if (value instanceof Collection) {
    136     136         Collection collection = (Collection) value;

```

1 dubbo-common/src/main/java/org/apache/dubbo/common/utils/PojoUtils.java

```

@@ -580,6 +580,7 @@ private static Method getSetterMethod(Class<?> cls, String property, Class<?> va
    580     580         for (Method m : cls.getMethods()) {
    581     581             if (ReflectUtils.isBeanPropertyWriteMethod(m) && m.getName().equals(name)) {
    582     582                 method = m;
    583     583     +                 break;
    584     584             }
    585     585         }

```

Figure 6 – Bug Fix in code smell snippets.

```

1109 private static void recalculateActivities(GPCalendarCalc calendar, Task task, List<TaskActivity> output, Date startDate,
1110     Date endDate) {
1111     TaskActivitiesAlgorithm alg = new TaskActivitiesAlgorithm(calendar);
1112     alg.recalculateActivities(task, output, startDate, endDate);
1113 }
1114

```

Figure 7 – Long Parameter List.

SAMARTHYAM; SHARMA, 2015a). Providing more access than required can expose implementation details to the client, and security issues that lead to bugs.

Broken Hierarchy. The form of this smell occurs usually is when the inheritance relationship between the supertype and its subtype is "inverted". In other words, the subtype is the generalization of the supertype instead of the other way around. When this happens, and the clients attempt to assign objects of subtype to a supertype references, they are exposed to undesirable or unexpected behaviors that can cause bugs in the application (SURYA-NARAYANA; SAMARTHYAM; SHARMA, 2015c).

Cyclic-Dependent Modularization. This smell arises when two or more abstractions depend of each other directly or indirectly (creating a tight coupling between the abstractions). In the presence of cyclic dependencies, the abstractions that are cyclically-dependent may need to be understood, changed, used, tested, or reused together. Further, in case of cyclic dependencies, changes in one class (say A) may lead to changes in other classes in the cycle (say B). However, because of the cyclic nature, changes in B can have ripple effects on the class where the change originated (i.e., A). Large and indirect cyclic dependencies are usually difficult to detect in complex software systems and are a common source of subtle bugs (SURYANARAYANA; SAMARTHYAM; SHARMA, 2015b).

Magic Number. The use of numbers directly in the code is considered Magic Number. The use of *magic numbers* in code obscures the developers' intent in choosing that number (MARTIN, 2008), increasing the bug proneness on changes in this parts of the code.

Summary of RQ₁. Code smell detection tools identify a large number of smells, but only a few of them are associated with bugs. The *Abstract Function Call From Constructor* presents the highest proportion of harmful code.

3.2 RQ₂. Do developers perceive and handler code smells as harmful?

Overall, 77 developers completed our survey. Figure 8 summarizes the survey results.

Developers' Perceptions. The vast majority of the developers (90.4%) consider code smells harmful to the software quality. In fact, 76.9% of the developers consider code smells very harmful. Some of them left some comments regarding the code smells harmfulness:

"Code smells often indicate or lead to bigger problems. Those bigger problems can make a code base fragile, difficult to maintain, and prone to errors."

"Code smells make software readability and comprehensibility worse. That itself already degrades software quality. Moreover, code smells can make it harder to find bugs in the software, since you can only find bugs in code that you can read, and understand."

Although developers believe that code smells are harmful, the results in RQ₁ indicate that

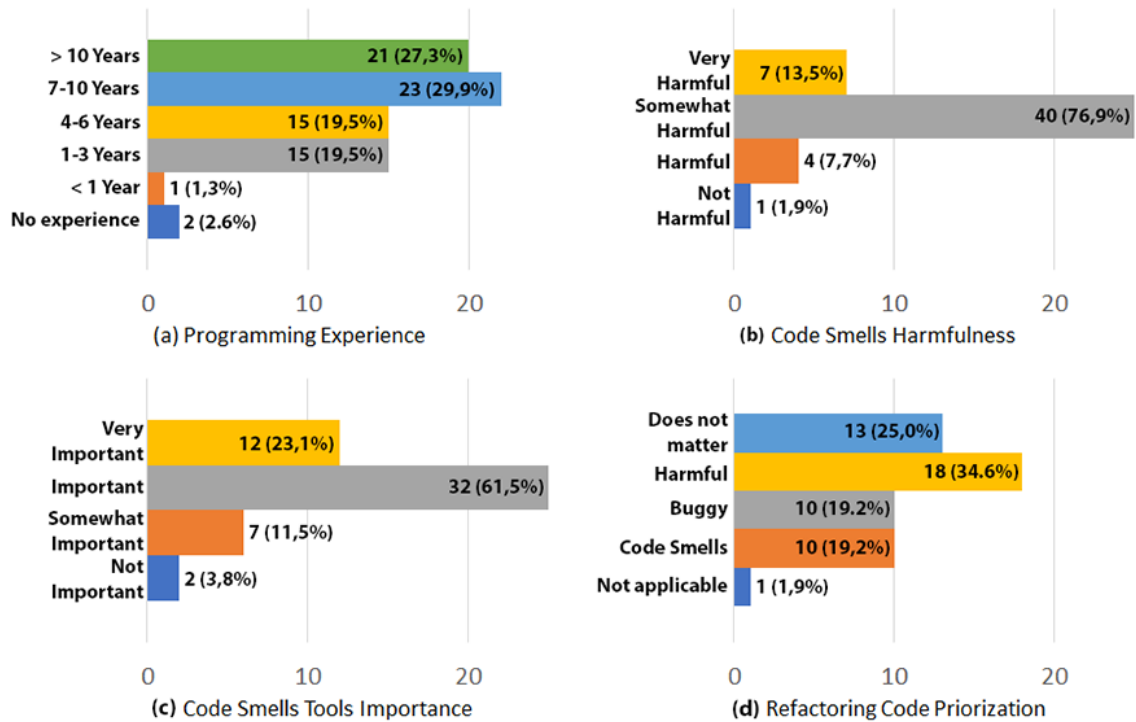


Figure 8 – Survey Results.

the vast majority of code smells may not be harmful to the software. This contradictory result introduces some questions that deserve to be discussed. Our study assesses the code smells harmfulness only based on bugs. The developers' comments suggest that they also consider other factors related to readability, maintainability, cost, and effort to fix it. Even though we only consider bugs in our analysis, they represent a determining factor for developers to refactor a code. Note that 53.8% of the developers prioritize code associated with bugs. Some comments of them help us to better understand this prioritization:

“When refactoring, it is ideal for dealing with all modules of the code, but due to some limitations, such as time, it is most important to look at fragile parts as harmful code.”

“Since a smell only indicates there might be a problem and harmful code definitely does have a problem, you should look at that first.”

Handling Code Smells. Our results indicate that developers can be convinced that code

smells are harmful to the software, but they may not be aware that the detection tools are not previously configured to detect smells that are really harmful. In fact, we observe that 84.6% of the developers believe that code smell detection tools are important. But, when we analyze the comments, there is no concern in selecting tools able to detect code smells that are really harmful to the software quality. Most of the developers mention code smell detection tools that we use in our study.

Summary of RQ₂. Most of the developers (90.4%) perceive code smell as harmful to the software, but they do not concern on selecting/customizing tools to detect harmful code.

3.3 RQ₃. How effective are Machine Learning techniques to classify harmful code?

In the previous section, we observed that even though code smells detection tools identify a large number of smells, only a few of them are really harmful. Hence, it is important to investigate whether ML-techniques classify harmful code as effective as they classify code smells.

Although we analyze 22 smell types in our study, we evaluate the effectiveness of the ML techniques in 11 types. The remaining smell types present a low number of bugs associated with them, as described in Table 4. Figures 9 and 10 present the effectiveness of the ML-techniques to recognize *Smelly* and *Harmful* code, respectively. In each figure, the y-axis describes the *f-measure* value reached by each technique on classifying the analyzed smell type. In addition, we attach the exact value to the bar associated with each technique.

HARMFUL. For the 11 smell types analyzed, the ML techniques could reach high effectiveness on classifying HARMFUL code, reaching an effectiveness of at least 97% in all the cases analyzed. In particular, **Random Forest reaches an effectiveness equal or greater than the other algorithms in ten of the 11 smell types. Also, Gradient Boosting present an effectiveness slightly lower than Random Forest**, reaching an effectiveness equal or greater than the other algorithms in nine smell types analyzed. Both techniques are more effective in all smells types, except in the *Deficient Encapsulation* smell type, where *Decision Tree* is more effective. On the other hand, **GaussianNB reaches the lowest effectiveness in six of the smell types** (*Magic Number*, *Insufficient Modularization*, *Unutilized Abstraction*,

Cyclic-Dependent Modularization, Deficient Encapsulation and Long Statement).

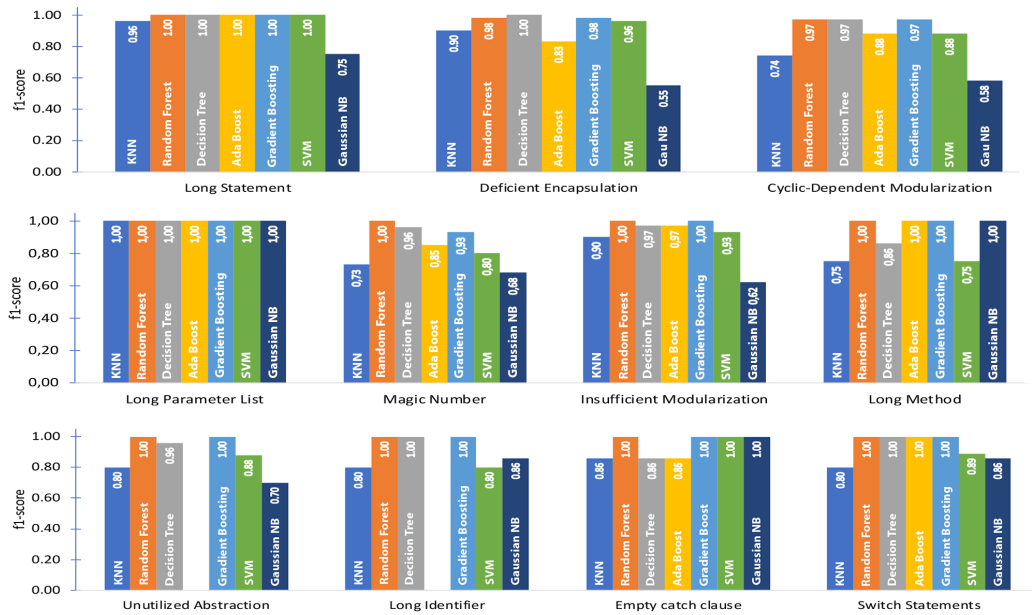


Figure 9 – Results for Harmful Code (f1-score).

SMELLY. Differently from HARMFUL code, the ML techniques could not reach a high effectiveness on classifying some smell types. While the ML techniques reach an effectiveness of at least 97% on classifying harmful code, these techniques could not reach an effectiveness greater than 75% in five of the 12 smell types analyzed (*Switch Statements, Long Identifier, Insufficient Modularization, Cyclic-Dependent Modularization and Deficient Encapsulation*). Similarly to HARMFUL code, *GaussianNB* presents a low effectiveness, reaching the lowest value in six of the smell types analyzed. On the other hand, *Gradient Boosting* is the most effective algorithms with an effectiveness equal or greater than *Random Forest* and *SVM* in six smell types (*Long Identifier, Long Parameter List, Unused Abstraction, Cyclic-Dependent Modularization, Long Method and Long Statement*). Both *Random Forest* and *SVM* reach the highest effectiveness in three smell types.

Summary of RQ₃. ML-techniques classify harmful code as effective as they identify code smells. While *Random Forest* and *Gradient Boosting* are effective on detecting both SMELLY and HARMFUL code, *GaussianNB* is the less effective technique.

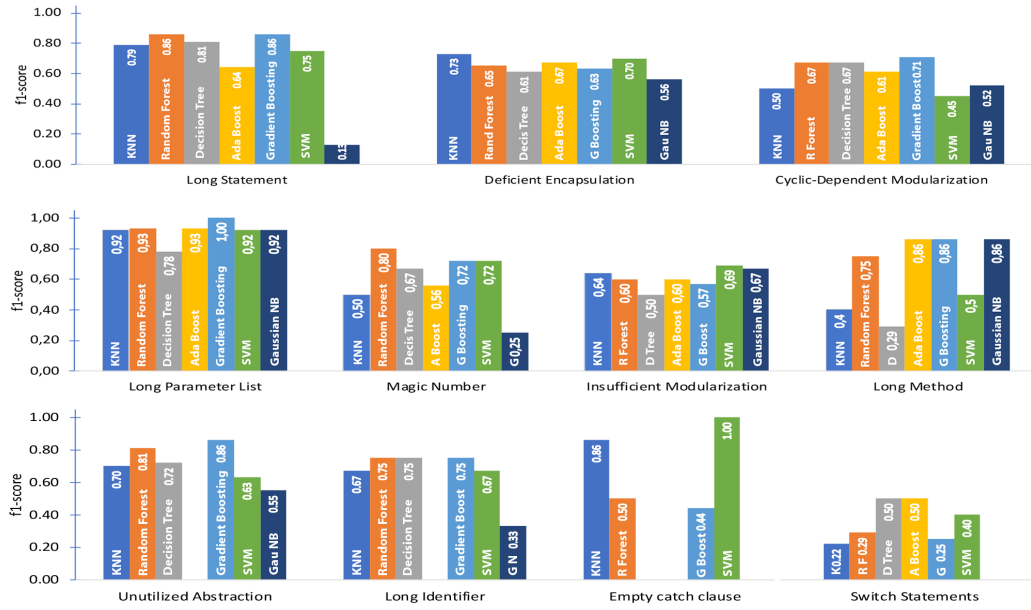


Figure 10 – Results for Smelly Code (f1-score).

3.4 RQ₄. Which metrics are most influential on classifying harmful code?

Figure 11 shows the results of the most important metrics in terms of *SHAP* values for each smell. In each figure, the y-axis describes the *SHAP* value reached by each metric on classifying harmful code according to the smell type analyzed. In addition, we attach the exact value to the bar associated with each metric.

Parameters Amount (PA). The *PA* metric has the highest importance on detecting harmful code in the *Long Statement* smell type, due the high number of parameters directly influence the Statements, becoming longer as the number of parameters increases. Reaching an importance of 0.1951. In the case of the *Deficient Encapsulation* and *Long Parameter List*, the *PA* metric has an importance slightly lower than *QUW* and *NQ*, reaching a value of 0.0709 and 0.1176 respectively.

Weight Method Class (WMC). The *WMC* metric has the highest importance on detecting harmful code in the smell types *Insufficient Modularization*, *Empty Catch Clause* and *Deficient Encapsulation*. It reaches an importance of 0.0446 in the *Insufficient Modularization* and 0.1467 in the *Empty Catch Clause*. In the case of the *Deficient Encapsulation*, the *PA* metric has an importance slightly lower than *WMC*, reaching a value of 0.0709. *WMC* is

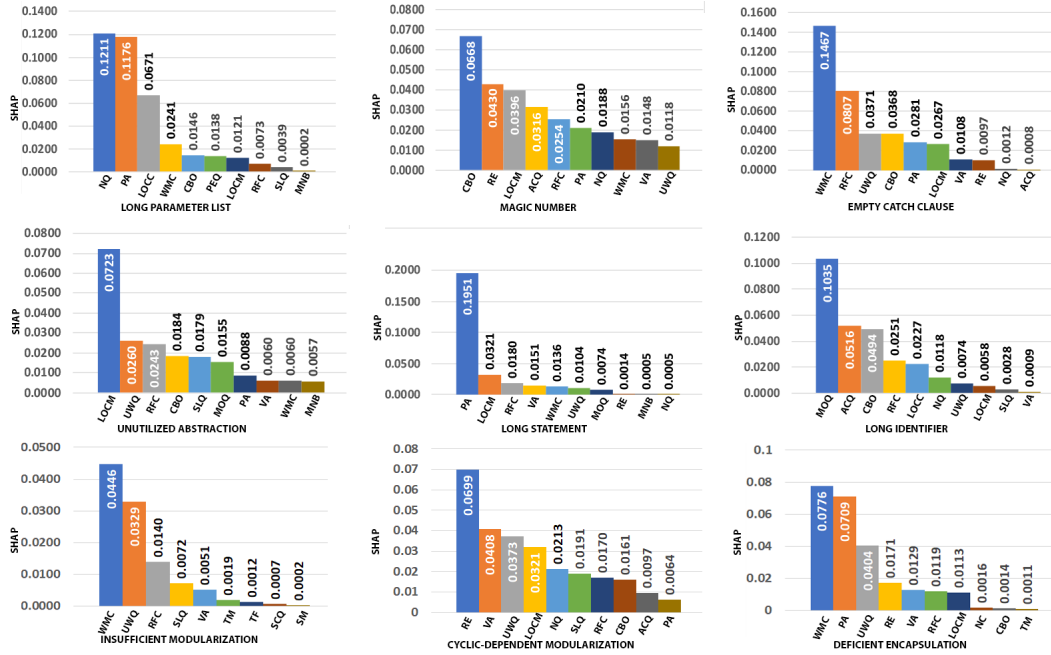


Figure 11 – Most Important metrics among code smells (SHAP values).

directly related to the bunge’s definition of complexity of a thing, since methods are properties of object classes and complexity is determined by the cardinality of its set of properties, the larger the number of methods in a class the greater the potential impact on children (Shyam R. Chidamber; Chris F. Kemerer, 1994), *insufficient modularization* and *deficient encapsulation* that involves more than one class making the influence of WMC higher.

Math Operations Quantity (MOQ), Numbers Quantity (NQ), and Couple Between Objects (CBO). These metrics reach the highest importance in *Long Identifier*, *Long Parameter List* and *Magic Number*. While *MOQ* reaches an importance of 0.1035, in the *Long Identifier*, *NQ* reaches an importance of 0.1211 in the *Long Parameter List*. In the case of *Magic Number*, only *CBO* could reach an importance above 0.05.

Returns (RE) and Lines of Code Method (LOCM). The *RE* and *LOCM* metrics present the highest importance in *Cyclic-Dependent Modularization* (0.0699) and *Unused Abstraction* (0.0723), respectively. *Assignments Qty* reaches the highest importance in the *Long Parameter List*, (0.159). Also, these metrics metric have the second highest importance in the small types *Magic Number* and *Long Statement*, reaching a value of 0.0430 and 0.0321 respectively. The cyclic-dependent modularization raises when two or more abstractions depend of each other (SURYANARAYANA; SAMARTHYAM; SHARMA, 2015b),

that dependency is directly related to the returns or no void methods, for e.g. the *ClassA* has methods (*methodA*, *methodB*) that returns values used by *methodC*, *methodD* of *ClasssB*.

Variables Amount (VA), Response for a Class (RFC), Anonymous Classes Quantity (ACQ), Unique Words Quantity (UWQ). The *VA*, *RFC* and *UWQ* reach the second highest importance in *Cyclic-Dependent Modularization*, *Empty Catch Clause* and *Long Identifier* with SHAP values of 0.0408, 0.0807 and 0.0516 respectively. Similar to these metrics, the *ACQ* also reaches a higher importance of 0.0329 and 0.0260 in the smell types *Insuficient Modularization* and *Unutilized Abstraction*.

Developer Metrics. Among the developer metrics analyzed in our study, only *Number of Commits (NC)* reaches some importance. In this case, this metric presents the importance of 0.0016 in the *Deficient Encapsulation*. Analyzing this case, we check that the developer that has a low number of commits in a project usually are directly involved with class that contains *Deficient Encapsulation*. The remaining metrics present importance below 0.001 in this smell type. One might argue that developer metrics are not important to detect harmful code since only one reaches some importance. However, note that we analyze the importance of 52 software metrics and only 11 developers' metrics. Even analyzing a greater number of software metrics, the developer ones could reach some importance in the *Deficient Encapsulation*. These results show that it is worthwhile to investigate developer metrics as other contexts in software engineering (i.e., bug prediction), but it is still early to affirm that they are useful since it is the first study that utilizes developer metrics in the context of code smells.

Summary of RQ₄. Software Metrics reach the highest importance. *Weight Method Class (WMC)* present the highest importance in three smells. *RE*, *MOQ*, *NQ*, *PA*, *CBO*, *LOCM* metrics reach the highest importance in the remaining smells. Developers metrics (such as *Number of Commits (NC)*) reach some importance in the smell *Deficient Encapsulation*.

4 Implications

The **harmful code, developers' perception, effectiveness** and the most **influential metrics**, provide valuable knowledge for researchers to extend the work and for the developers provides information about which smell types they should put more attention on the refactoring process.

Harmful Code. Existing studies have focused their researches on the impact of code smells on the change and fault-proneness. Our results confirm the results of those studies, and can be useful for researchers to further investigate and explore the relation between some smell types and bugs, e. g., why switch statements are more harmful than long methods?.

Developers' Perception. Our survey results show that most of the developers perceive code smell as harmful to the software. Thus, we can create new tools to help them on smells identification and refactoring.

Effectiveness. Our results show that machine learning techniques classify harmful code as effective as they identify code smells. Developers and Researchers will be able to use those techniques to help them in future works involving harmful code.

Influential Metrics. Our results show that Software Metrics reach the highest importance, and only one developer metric (*Number of Commits (NC)*) reached some importance, however, we analyzed only 11 developers metrics against 52 software metrics, and even analyzing a greater number of software metrics, the developer could some importance. This confirms that it is worthwhile to investigate developer metrics as other contexts in software engineering (i.e., bug prediction).

5 Related Work

This chapter includes the related work performed on **Impact, Detection Tools and Techniques, Developers' Perceptions**, which provided inspiration for our study.

Impact. Recent studies (PALOMBA et al., 2018; MA et al., 2018) investigated the impact of code smells in the change and fault-proneness. Palomba et al. (PALOMBA et al., 2018) conduct a study on 395 releases of 30 open-source projects and considering 17,350 code smells manually validated of 13 different types. The results show that classes with code smells have a higher change-and-fault-proneness than smell-free classes. Moreover, another study (MA et al., 2018) investigated the relationship between smells and fine-grained structural change-proneness. They found that, in most cases, smelly classes are more likely to undergo structural changes.

Tracy et al. (HALL et al., 2014) investigated the relationship between faults and five smells: Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man. They collected the fault data from changes and faults in the repositories of the systems. Their findings suggest that some smells do indicate fault-prone in some circumstances, and they have different effects on different systems.

Besides, all these studies provide evidence that bad code smells have negative effects on some maintenance properties. Our study complements them on pointing the smells that are really harmful to the software, helping developers prioritize them while refactoring the code.

Detection Tools and Techniques. Previous studies (KHOMH et al., 2009; KHOMH et al., 2011; Maiga et al., 2012; AMORIM et al., 2016; Arcelli Fontana et al., 2016) have indicated the use of machine learning techniques as a promising way to detect code smells. These techniques use examples of code smells previously reported by the developer in order to learn how to detect such smells.

In this way, Amorim et al. (AMORIM et al., 2016) present a preliminary study on the effectiveness of decision tree techniques to detect code smells by using four open-source projects. The results indicate that this technique can be effective to detect code smells. Other studies (Maiga et al., 2012; KHOMH et al., 2009) also inferred ML techniques to detect code smells by using Bayesian Belief Networks (KHOMH et al., 2009) or Support-vector machine (Maiga et al., 2012).

Hozano et al. (HOZANO et al., 2017) introduced *Histrategy*, a guided customization technique to improve the efficiency on smell detection. *Histrategy* considers a limited set of detection strategies, produced from different detection heuristics, an input of a customization process. The output of the customization process consists of a detection strategy tailored to each developer. The technique was evaluated in an experimental study with 48 developers and four types of code smell. The results show that *Histrategy* is able to outperform six widely adopted machine learning algorithms – used in unguided approaches – both in effectiveness and efficiency.

Fontana et al. (Arcelli Fontana; ZANONI, 2017) extended their previous work (Arcelli Fontana et al., 2016) by applying several machine learning techniques, varying from multinomial classification to regression. In this study, the authors modeled the code smell harmfulness as an ordinal variable and compared the accuracy of the techniques.

Even though the previous studies (KHOMH et al., 2009; KHOMH et al., 2011; Maiga et al., 2012; AMORIM et al., 2016; Arcelli Fontana et al., 2016) have contributed to evidence the effectiveness of ML-techniques in the detection of code smells, none of these studies analyze how effective are these techniques to detect harmful code. Our study complements all these approaches. Specifically, we focus on the identification of *harmful code*, supported by the addition of new features (Developers' Metrics and *Bugs*) approach similar to Catolino et al. work (CATOLINO et al., 2019).

Developers' Perceptions. Palomba et al. (PALOMBA et al., 2014) conducted a survey to investigate developers' perception on bad smells, they showed to developers code entities affected and not by bad smells, and asked them to indicate whether the code contains a potential design problem, nature, and severity. The results of their study distill the following lessons learned: I. There are some smells that are generally not perceived by developers as design problems. II. The instance of a bad smell may or may not represent a problem based on the “intensity” of the problem. III. Smells related to complex/long source code are generally perceived as an important threat by developers. IV. Developer's experience and system's knowledge pay an important role in the identification of some smells.

Sae-Lim et al. (SAE-LIM; HAYASHI; SAEKI, 2017) investigated professional developers to determine the factors that they use for selecting and prioritizing code smells. They found that *Task Relevance* and *Smell Severity* were most commonly considered during code

smell selection, while *Module Importance* is employed most often for code smell selection.

Hozano et al. (HOZANO et al., 2018) performed a broader study to investigate how similar developers detect code smells. They conducted an empirical study with 75 developers who evaluated instances of 15 different code smells types. For each smell type, they analyzed the agreement among developers. Their results indicated that the developers presented a low agreement on detecting all 15 smell types analyzed, also factors related to background and experience did not have a consistent influence on the agreement among the developers.

Our study supports those previous studies. Our survey results confirm their results on suggesting that developers consider factors related to readability, maintainability, cost, and effort to fix while detecting smells.

6 Threats to Validity

In this chapter, we present the threats to validity by following the (WOHLIN et al., 2012) validity criteria.

Construct Validity. In our study, we collected a set of code smells that were not manually validated. To mitigate this threat, we used tools and configurations used in previous studies (HOZANO et al., 2017; HOZANO et al., 2018; FONTANA et al., 2011; FONTANA; BRAIONE; ZANONI, 2012; Arcelli Fontana et al., 2016; Arcelli Fontana; ZANONI, 2017; AMORIM et al., 2016; ZAIDMAN et al., 2017). Another threat to validity is to identify commits that fixed bugs correctly. GitHub provides the functionality to close issues by commits messages or pull requests comments. We mitigated this threat by identifying as the bug-fix, the commits, or pull requests (the last commit) that close issues labeled as “bug” or “defect” using this functionality. Besides, we identified methods and classes associated with each *bug* and establish the immediate previous commit of these methods and classes as buggy code. However, some buggy methods and classes could not have a bug directly associated with him since developers may be working on code improvements or new features during a fix. Also, some types of harmful code may not be identified because the tools used in the research not identify all the types of code smells.

Internal Validity. Another threat is the procedures of the steps adopted in Section 2.9. These steps are related to the type of dataset splitting, selection of hyper-parameters, and construction of the ML-techniques. To mitigate this, we relied on decisions made by previous studies that obtained good results detecting code smells using ML-techniques (AZEEM et al., 2019)

External Validity. Regarding the validity of our findings, we selected only projects in which the primary language adopted is *Java*. Although we have selected a large number of projects from six different domains with different sizes and developers, our results might not be generalized to other projects which *Java* is not the primary language as those projects may have different characteristics.

7 Conclusions and Future Works

This chapter will draw conclusions based on the results, discussion, and contribution of our work. Lastly, some ideas for future work will be discussed.

We presented a study to understand and classify code harmfulness. First, we analyzed the occurrence of CLEAN, SMELLY, BUGGY, and HARMFUL code in open-source projects as well as which smell types are more related to harmful code. Further, we investigated the developers' perceptions regarding the harmfulness of code smells. We also evaluated the effectiveness of machine learning techniques to detect harmful and smelly code. Finally, we investigated which metrics are most important in harmful code detection.

To perform our study, we defined an experiment with 22 smell types, 803 versions of 12 open-source projects, 40,340 bugs mined from GitHub issues and 132,219 code smells. The results show that even though we have a high number of code smells, only 0.07% of those smells are harmful. The *Abstract Function Call From Constructor* is the smell type more related to harmful code. Also, we performed a survey with 77 developers to investigate their perceptions regarding the harmfulness of code smells. Most of them (90.4%) consider code smells harmful to the software, and 84.6% of those developers believe that code smells detection tools are important. However, those developers do not concern about selecting tools able to detect harmful code. Regarding the effectiveness of machine learning techniques to detect harmful code, our results indicate that they reach effectiveness at least 97%. While the **Random Forest** (HO, 1995) is effective in detecting both smelly and harmful code, the **Gaussian Naive Bayes** (HAND; YU, 2001) is the less effective technique. Finally, our results suggest software metrics (such as *CBO (Couple Between Objects)* and *WMC (Weight Method Class)*) are important to customize machine learning techniques to detect harmful code.

As future work, we intend to extend this investigation by adding more projects, evaluating the efficiency of harmful code using different machine learning techniques such as multi-layer perceptron, and introduce change proneness features to improve the harmful code detection.

Bibliography

ABBES, M. et al. An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR, IEEE*, p. 181–190, 2011. ISSN 15345351.

ALENEZI, M.; ZAROOUR, M. An empirical study of bad smells during software evolution using designite tool. *i-Manager's Journal on Software Engineering*, v. 12, n. 4, p. 12–27, Apr 2018. Disponível em: <<https://search.proquest.com/docview/2148827386?accountid=26580>>.

ALTMAN, N. S. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, [American Statistical Association, Taylor Francis, Ltd.], v. 46, n. 3, p. 175–185, 1992. ISSN 00031305. Disponível em: <<http://www.jstor.org/stable/2685209>>.

AMORIM, L. et al. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. *2015 IEEE 26th International Symposium on Software Reliability Engineering, ISSRE 2015, IEEE*, p. 261–269, 2016.

Arcelli Fontana, F. et al. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering, Empirical Software Engineering*, v. 21, n. 3, p. 1143–1191, 2016. ISSN 15737616. Disponível em: <<http://dx.doi.org/10.1007/s10664-015-9378-4>>.

Arcelli Fontana, F.; ZANONI, M. Code smell severity classification using machine learning techniques. *Knowledge-Based Systems, Elsevier B.V.*, v. 128, p. 43–58, 2017. ISSN 09507051.

AZEEM, M. I. et al. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology, Elsevier B.V.*, v. 108, n. 4, p. 115–138, 2019. ISSN 09505849. Disponível em: <<https://doi.org/10.1016/j.infsof.2018.12.009>>.

BERGSTRA, J.; BENGIO, Y. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, JMLR.org, v. 13, n. 1, p. 281–305, fev. 2012. ISSN 1532-4435. Disponível em: <<http://dl.acm.org/citation.cfm?id=2503308.2188395>>.

BERGSTRA, J.; YAMINS, D.; COX, D. D. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In: *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*. JMLR.org, 2013. (ICML'13), p. I–115–I–123. Disponível em: <<http://dl.acm.org/citation.cfm?id=3042817.3042832>>.

BOSER, B. E.; GUYON, I. M.; VAPNIK, V. N. A training algorithm for optimal margin classifiers. In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. New York, NY, USA: ACM, 1992. (COLT '92), p. 144–152. ISBN 0-89791-497-X. Disponível em: <<http://doi.acm.org/10.1145/130385.130401>>.

BREIMAN, L. (34) Classification and regression trees Regression trees. *Encyclopedia of Ecology*, v. 40, n. 3, p. 582–588, 1984. ISSN 1661-8564.

BREIMAN, L. et al. *Classification and regression trees*. [S.l.]: Wadsworth Publishing Company, 1984.

CATOLINO, G. et al. Improving change prediction models with code smell-related information. *Empirical Software Engineering*, 2019.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, IEEE Press, Piscataway, NJ, USA, v. 20, n. 6, p. 476–493, jun. 1994. ISSN 0098-5589. Disponível em: <<http://dx.doi.org/10.1109/32.295895>>.

DESIGNITE Site. 2020. Disponível em: <<http://tusharma.in/smells/>>.

Di Nucci, D. et al. Detecting code smells using machine learning techniques: Are we there yet? In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.: s.n.], 2018. p. 612–621.

FALCÃO, F. et al. Influence of Technical and Social Factors for Introducing Bugs. 2018. Disponível em: <<http://arxiv.org/abs/1811.03758>>.

FONTANA, F. A.; BRAIONE, P.; ZANONI, M. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, v. 11, n. 2, 2012. ISSN 16601769.

FONTANA, F. A. et al. On experimenting refactoring tools to remove code smells. In: *Scientific Workshop Proceedings of the XP2015*. New York, NY, USA: ACM, 2015. (XP '15 workshops), p. 7:1–7:8. ISBN 978-1-4503-3409-9. Disponível em: <<http://doi.acm.org/10.1145/2764979.2764986>>.

FONTANA, F. A. et al. An experience report on using code smells detection tools. *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2011*, p. 450–457, 2011.

FOWLER, M. *Refactoring: improving the design of existing code*. [S.l.]: Addison-Wesley, 1999.

GRIFFOR, E. et al. *Framework for Cyber-Physical Systems: Volume 1, Overview*. 2017.

GUEHENEUC, Y.-G.; SAHRAOUI, H.; ZAIDI, F. Fingerprinting design patterns. In: *Proceedings of the 11th Working Conference on Reverse Engineering*. USA: IEEE Computer Society, 2004. (WCRE '04), p. 172–181. ISBN 0769522432.

HALL, T. et al. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology*, v. 23, n. 4, 2014. ISSN 15577392.

HAND, D. J.; YU, K. Idiot's bayes: Not so stupid after all? *International Statistical Review / Revue Internationale de Statistique*, [Wiley, International Statistical Institute (ISI)], v. 69, n. 3, p. 385–398, 2001. ISSN 03067734, 17515823. Disponível em: <<http://www.jstor.org/stable/1403452>>.

HARMFUL Code. 2020. Disponível em: <<https://harmfulcode.github.io/>>.

HO, T. K. Random decision forests. In: *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*. Washington, DC, USA: IEEE Computer Society, 1995. (ICDAR '95), p. 278–. ISBN 0-8186-7128-9. Disponível em: <<http://dl.acm.org/citation.cfm?id=844379.844681>>.

HOZANO, M. et al. Smells Are Sensitive to Developers! on the Efficiency of (Un)Guided Customized Detection. *IEEE International Conference on Program Comprehension*, IEEE, p. 110–120, 2017.

HOZANO, M. et al. Are you smelling it? Investigating how similar developers detect code smells. *Information and Software Technology*, Elsevier B.V., v. 93, p. 130–146, 2018. ISSN 09505849. Disponível em: <<https://doi.org/10.1016/j.infsof.2017.09.002>>.

KESSENTINI, M. et al. Search-based design defects detection by example. In: *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice of Software*. Berlin, Heidelberg: Springer-Verlag, 2011. (FASE'11/ETAPS'11), p. 401–415. ISBN 978-3-642-19810-6. Disponível em: <<http://dl.acm.org/citation.cfm?id=1987434.1987471>>.

KHOMH, F. et al. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, v. 17, n. 3, p. 243–275, 2012. ISSN 13823256.

KHOMH, F. et al. A bayesian approach for the detection of code and design smells. *Proceedings - International Conference on Quality Software*, p. 305–314, 2009. ISSN 15506002.

KHOMH, F. et al. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, Elsevier Inc., v. 84, n. 4, p. 559–572, 2011. ISSN 01641212. Disponível em: <<http://dx.doi.org/10.1016/j.jss.2010.11.921>>.

KIM, S. et al. Automatic identification of bug-introducing changes. In: *IEEE. Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. [S.l.], 2006. p. 81–90.

KOMER, B.; BERGSTRA, J.; ELIASMITH, C. Hyperopt-sklearn: Automatic hyperparameter configuration for scikit-learn. In: . [S.l.: s.n.], 2014. p. 32–37.

LANZA, M.; MARINESCU, R. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 3642063748, 9783642063749.

LUNDBERG, S. M.; LEE, S.-I. A unified approach to interpreting model predictions. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. [S.l.]: Curran Associates Inc., 2017. (NIPS'17), p. 4768–4777. ISBN 978-1-5108-6096-4.

MA, W. et al. Exploring the Impact of Code Smells on Fine-Grained Structural Change-Proneness. *International Journal of Software Engineering and Knowledge Engineering*, v. 28, n. 10, p. 1487–1516, 2018. ISSN 0218-1940.

MACIA, I. et al. On the relevance of code anomalies for identifying architecture degradation symptoms. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR, IEEE*, p. 277–286, 2012. ISSN 15345351.

Maiga, A. et al. Support vector machines for anti-pattern detection. In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.: s.n.], 2012. p. 278–281.

MARINESCU, R. Detecting Design Flaws via Metrics in Object-Oriented Systems A Metrics-Based Approach for Problem Detection. *International Conference and Technology of Object-Oriented Languages and Systems (TOOLS)*, p. 173–182, 2001.

Marinescu, R. Detection strategies: metrics-based rules for detecting design flaws. In: *20th IEEE International Conference on Software Maintenance, 2004. Proceedings*. [S.l.: s.n.], 2004. p. 350–359. ISSN 1063-6773.

MARTIN, R. Clean code: A handbook of agile software craftsmanship. 01 2008.

MUNRO, M. J. Product metrics for automatic identification of “bad smell” design problems in Java source-code. *Proceedings - International Software Metrics Symposium*, v. 2005, n. Metrics, p. 125–133, 2005. ISSN 15301435.

OLBRICH, S. M.; CRUZES, D. S.; SJOØBERG, D. I. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. *IEEE International Conference on Software Maintenance, ICSM*, 2010.

PALOMBA, F. et al. Detecting bad smells in source code using change history information. *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, IEEE, p. 268–278, 2013.

PALOMBA, F. et al. Do they really smell bad? A study on developers’ perception of bad code smells. *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014*, IEEE, p. 101–110, 2014.

PALOMBA, F. et al. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, IEEE, v. 41, n. 5, p. 462–489, 2015. ISSN 00985589.

Palomba, F. et al. Do they really smell bad? a study on developers’ perception of bad code smells. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. [S.l.: s.n.], 2014. p. 101–110. ISSN 1063-6773.

PALOMBA, F. et al. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, Empirical Software Engineering, v. 23, n. 3, p. 1188–1221, 2018. ISSN 15737616.

PEDREGOSA, F. et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, v. 12, p. 2825–2830, 2011.

PMD Site. 2020. Disponível em: <https://pmd.github.io/latest/pmd_rules_java_design.html>.

SAE-LIM, N.; HAYASHI, S.; SAEKI, M. How do developers select and prioritize code smells? A preliminary study. *Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017*, p. 484–488, 2017.

SANTOS, F. F. ao B. D. et al. On relating technical, social factors, and the introduction of bugs. In: *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*. [S.l.: s.n.], 2020. p. To appear.

SCHAPIRE, R. E. A brief introduction to boosting. In: *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. (IJCAI'99), p. 1401–1406. Disponível em: <<http://dl.acm.org/citation.cfm?id=1624312.1624417>>.

SHARMA, T.; MISHRA, P.; TIWARI, R. Designite: A software design quality assessment tool. In: *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*. New York, NY, USA: ACM, 2016. (BRIDGE '16), p. 1–4. ISBN 978-1-4503-4153-0. Disponível em: <<http://doi.acm.org/10.1145/2896935.2896938>>.

Shyam R. Chidamber; Chris F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, v. 20, n. 6, p. 476 — 493, 1994.

SPADINI, D.; ANICHE, M.; BACCHELLI, A. PyDriller: Python framework for mining software repositories. *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, p. 908–911, 2018.

SURYANARAYANA, G.; SAMARTHYAM, G.; SHARMA, T. Chapter 4 - encapsulation smells. In: SURYANARAYANA, G.; SAMARTHYAM, G.; SHARMA, T. (Ed.). *Refactoring for Software Design Smells*. Boston: Morgan Kaufmann, 2015. p. 61 – 91. ISBN 978-0-12-801397-7. Disponível em: <<http://www.sciencedirect.com/science/article/pii/B9780128013977000047>>.

SURYANARAYANA, G.; SAMARTHYAM, G.; SHARMA, T. Chapter 5 - modularization smells. In: SURYANARAYANA, G.; SAMARTHYAM, G.; SHARMA, T. (Ed.). *Refactoring for Software Design Smells*. Boston: Morgan Kaufmann, 2015. p. 93 – 122. ISBN 978-0-12-801397-7. Disponível em: <<http://www.sciencedirect.com/science/article/pii/B9780128013977000059>>.

SURYANARAYANA, G.; SAMARTHYAM, G.; SHARMA, T. Chapter 6 - hierarchy smells. In: SURYANARAYANA, G.; SAMARTHYAM, G.; SHARMA, T. (Ed.). *Refactoring for Software Design Smells*. Boston: Morgan Kaufmann, 2015. p. 123 – 192. ISBN 978-0-12-801397-7. Disponível em: <<http://www.sciencedirect.com/science/article/pii/B9780128013977000060>>.

SURYANARAYANA, G.; SAMARTHYAM, G.; SHARMA, T. Refactoring for software design smells. In: SURYANARAYANA, G.; SAMARTHYAM, G.; SHARMA, T. (Ed.).

Refactoring for Software Design Smells. Boston: Morgan Kaufmann, 2015. ISBN 978-0-12-801397-7. Disponível em: <<http://www.sciencedirect.com/science/article/pii/B9780128013977000047>>.

WILLIAMS, C.; SPACCO, J. Szz revisited: verifying when changes induce fixes. In: ACM. *Proceedings of the 2008 workshop on Defects in large software systems*. [S.l.], 2008. p. 32–36.

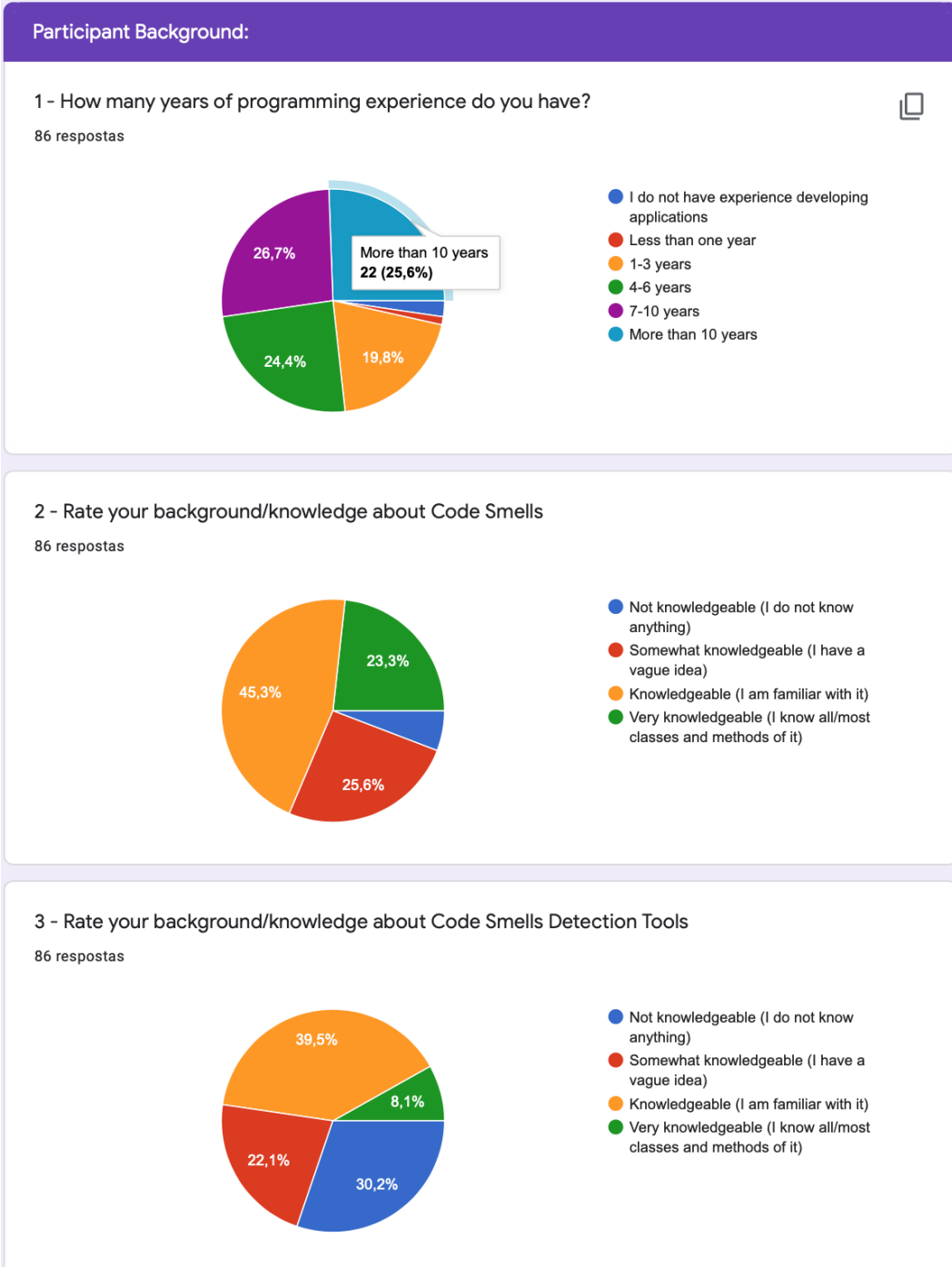
WOHLIN, C. et al. *Experimentation in Software Engineering*. [S.l.]: Springer Publishing Company, Incorporated, 2012. ISBN 3642290434, 9783642290435.

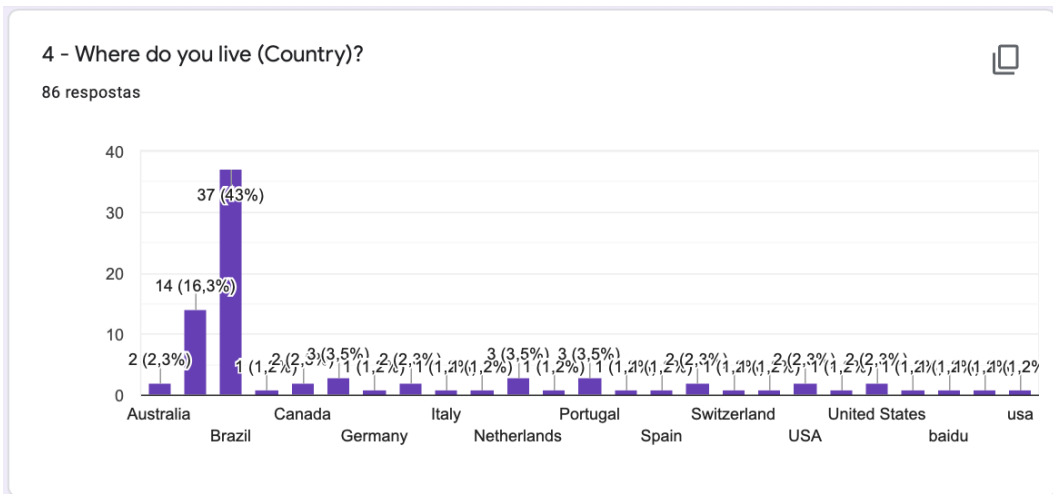
YAMASHITA, A.; MOONEN, L. Exploring the impact of inter-smell relations on software maintainability: An empirical study. *Proceedings - International Conference on Software Engineering*, IEEE, p. 682–691, 2013. ISSN 02705257.

ZAIDMAN, A. et al. The Scent of a Smell: An Extensive Comparison Between Textual and Structural Smells. *IEEE Transactions on Software Engineering*, IEEE, v. 44, n. 10, p. 977–1000, 2017. ISSN 0098-5589.

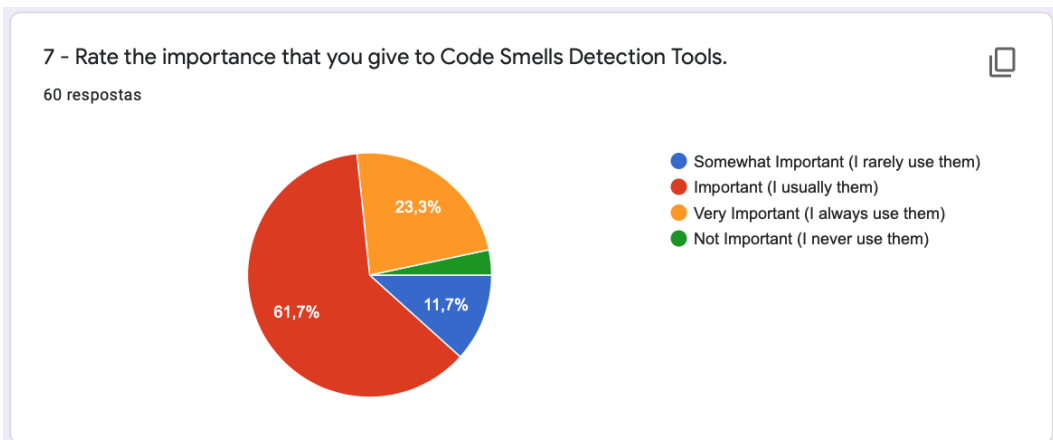
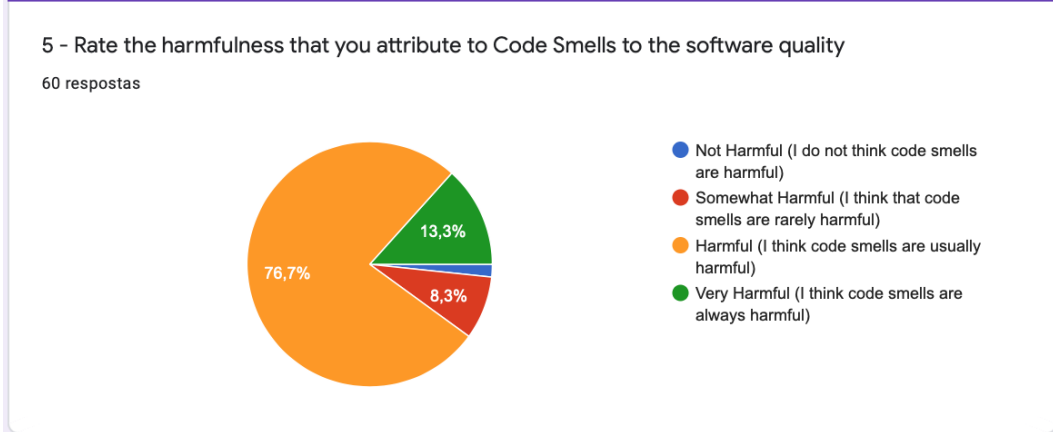
APPENDIX A – Supplementary Material

1 Survey





Code Smells Harmfulness



6 - Could you please justify your answer to question 5?

Code smells often indicate or lead to bigger problems. Those bigger problems can make a code base fragile difficult to maintain

It makes it difficult to read and maintain your code base.

It difficulties the readability of the code that can lead to the introductions of bugs.

They tend to indicate over complication or lacking SOLID principals.

Code smells have impacts on code maintainability when the code is not clear what should be done. This makes the misinterpretation of what the code should do and if there is no automated testing to guide the developer on what the code should do we often have the rework to rewrite all the functionality because it is simpler to develop from scratch. that improve a code snippet.

It can impact very badly in the long run of software development causing degraation of code quality.

Code smell is a way to detect bad decisions that can impact directly the runtime and the business around the software.

This can hardly be evaluated out of context. Some harmfulness is caused by the broken window effect anyway.

Smells are indications something MIGHT be wrong and it's LIKELY that you are violating some aspect of good OO design. But then again maybe not.

It usually impacts on maintainability the presence of code smells when the time of modify the existing code arrives code smells make that changes easy to break the code. I don't mark this as very harmful because it is preferable a well tested code with some code smells that a non tested software without code smells.

Code smell is an item for have a technical debt. Miss unit test is another item being harmful.

It will have direct consequences in quality maintenance if the smell is not fixed.

In a common basis we work with scalable and maintainable software, so any code smell probably means, in future, a huge refactor. There are worse code smells like the blob and stuffs, but any of them I consider a problem.

They always represent a relevant risk to the maintenance and, most of them, to future implementations, demanding a lot of time with refactor work.

If this problem isn't dealt with it'll spread, the cost and effort to fix it'll increase.

At the first moment this wouldn't a big problem but in the future it will cause problems to maintain the application.

It usually means you are working with people who don't care about the quality of the work or is in a unconscious incompetence level which means that they think they know what they are doing but they are clueless.

They increase the complexity of the codebase and the cognitive load on engineers. This leads to decreased productivity and increased bugs.

Code smells make software readability and comprehensibility worse. That itself already degrade software quality. Moreover code smells can make it harder to find bugs in the software since you can only find bugs in code that you can read and understand.

Code smells tend to make code understanding difficult and software maintenance very difficult.

Your code is going to be confusing to read.

Long methods require more time for maintenance and many times require devs with more experience.

Code smells usually indicates that something is wrong. When you have for example long methods code readability decreases and therefore system maintenance is adversely affected.

Complex and long methods are difficult to manage. I like to split complexity in smaller chunks.

I do believe that code smells are harmful but not usually harmful. There are specific cases that make them harmful for example when specific types of smells (or from similar categories) co-occur together. For instance if a class contains a God Class Intensive Coupling (or Dispersed Coupling) and more than one Feature Envy in this case the smells are harmful since they indicate that the class has a bigger structural problem. On the other hand there are cases that smells are not harmful for example when a class has Lazy Class.

I believe that if you have unit test around it it'd be Harmful or even Somewhat Harmful. Usually code smell is easy to be refactored when you have unit test.

If I need to improve I can introduce a bug.

A piece of bad code tend to degrade fast after a lot of time without proper refactoring/maintenance leading to a costly code evolution in the future.

I don't think that there are a mandatory relation between code smells and the harmfulness. But naturally in some cases code smells could contains characteristics of harmful code.

Some code can be ugly but it works.

Not all smells are so harmful.

It is difficult to introduce changes in the business logic.

The presence of code smell could increase code maintenance work.

No.

Code smell instances eventually hinder some major development tasks especially when it comes to maintaining and evolving systems. I've had a hard time to read and change some large classes complex methods and intricate hierarchies as well.

Hard maintenance.

8 - Could you please describe Code Smells Detection tools that you have used?

Rubocop

Sonarqube

It is not always in the development of the program to give yourself a macro view of it so it is likely that some code smell will not be identified.

Tools for detecting code smells are important because humans are subject to failure

After long years of experience I directly avoid all possible smells. For what I don't see at the first time there will be a code refactory.

PMD

Rubocop

At some time of de development I consider checking all the project in order to find possible code smells. The problem is still the lacking of good software tools to do so.

Sonarqube and Sonarlint.

Sonar

Sonar.

I am not currently using a tool to detect bad smells. I already used CheckStyle PMD and FindBugs but for academic purposes only. I believe that when Eclipse or Android Studio says there are unused variables or methods, then there might be something wrong with the code. So today I just use the traditional IDE warnings. Not a specific tool.

lints, code formatters, code analyzers.

Idea, SonarQube, Js/TsLint are the ones i feel more comfortable.

Sonarqube.

Sonar, pmd, owasp, findbugs, vpav

FindBugs, PMD, SonarQube.

Internal tools to validate methods sizes nested logic that can be potentially breakout in small chunks. Linters, code sniffers...

Static code analyzers

Sonar and ESLint

JDeodorant, JSPrIT, PMD

I've used in my career some tools to help me with code smells, like Checkstyle and more recently, we use Sonar.

Reek

PMD

Analyze Inspect Code - Android Studio

Sonar, Findbugs, jacoco

Most of them were used as part of a CI solution , like sonar, and works pointing out code smells during the development process and suggesting improvements to solve them.

I've published a literature review on these tools at EASE 2016 ("A review-based comparative study of bad smell detection tools"). Part of my work was using tools like PMD, inFusion, JDeodorant, and etc. I've also used some of these tools to detect smells for study purposes. But, to be honest, mostly I've written my own detection scripts to run them on spreadsheets with code metrics data... LOL

None

For Java I've used PMD and since I use IntelliJ IDEA , it has embedded static code analysis in it.

PMD and JDeodorant

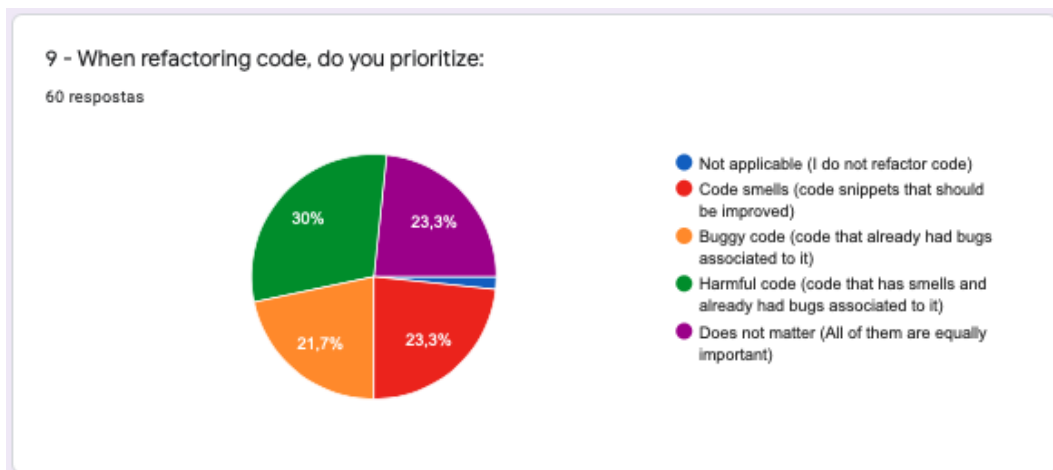
In the past I have used some tools such as PMD and JDeodorant. In both cases, the tools were used in non-graphical ways, just for research purposes.

No.

Linters in general

I use one that has been developed in the research group that I work with. The name is Organic 2.0

I have read research on automatic code smell detection , but I don't recall any tool names.



10 - Could you please justify your answer to question 9?

Since a smell only indicates there might be a problem and harmful code definitely does have a problem, you should look at that first.

Always leave the code better than before

We usually run sonarqube reports and try to fix all the warnings that are found.

Codes that allow bugs are problematic because they are harmful to execution.

You may have code that has code smells, but if they have good automated testing, refactoring becomes a simpler and less risky task. And there are different code smells, some impact the readability of implementation details more, if the public interface of the code is well written, but the internal code is poorly made, refactoring becomes even simpler. The problem is more in code than badly done, do not work. We should prioritize covering them with tests, then adjust them as guided by the tests discovering current code problems, then refactoring them.

In the real life we have to stay into the budget: so when, for whatever reason, I put my hand on a piece of code I try to fix/improve it.

I would fix all of it but time is limited so priorities have to be set.

All information is important.

I always consider improving the code quality when fixing a bug.

When touching on working code the risk to ruin something is greater than fixing already buggy code.

because if you are refactoring you are trying to improve the code. Does not matter if it's horrible or somewhat acceptable.

Code blocks that already have a bug and have code smell is much more problematic than a code smell.

Code smells should be found by static analysis tools or reviewer.

I usually use RENAME refactoring, so I'm interested in making the code easier to understand. Generally it's not about bug, but readability.

Economics. Buggy code have hard effects on end user which tend to abandon your product after working for some time with a bug.

The question is really tricky because you will refactor always the worst part, otherwise the code will not work. I see what do you want to achieve with that question but the way you are asking that is really biased.

Following the criteria will be harmful.

Short term business value. Smells are best addressed by coaching and boy scout approaches.

I refactor to improve reusability and reduce fragile or error-prone code. How the code got to that state is irrelevant.

Because at the company I work prioritize quality over delivery.

Any improvements are worth the time.

I normally refactoring a source code to improve them. I do not expect to catch a bug to refactoring the source code.

The priority is to fix bugs.

When my team works with software maintenance, we usually get a problem and start reviewing it. Usually this problem represents a bug; therefore, we have the opportunity to improve some parts of the code.

Refactoring process could insert new bugs, so I avoid to refactor code that it is working.

When refactor source code, we focus on long and complex codes, to facilitate future maintenance or upgrades.

I usually look for problems that already exist or might become one in the future.

Not all the problems are solved in a refactoring session, bug is your mission to deliver the best in that situation.

Working on code smells means working on the prevention of most of bugs that could appear, so worry only about those who already have bugs attached too it is never enough.

My academic background made me see refactorings as means to "clean the house" in terms of bad code structures rather than means to actually fix bugs. Eventually, refactorings can help fix a bug. Nevertheless, as a practitioner, I've tried the most to avoid refactoring buggy code elements. These elements should be changed only if necessary; otherwise, the damage can get worse (other bugs may emerge).

There is no priority.

I usually give priority to duplication, shotgun surgery, large class, naming, long methods, too many params.

Buggy code is something that already problematic to the software, so should be prioritized.

When refactoring, it is ideal to deal with all modules of the code, but due to some limitations, such as time, it is most important to look at fragile parts as harmful code.

Well written code avoids Code smells.

In general, I tend to fix the bug (first) as fast as possible, which means I do not think about quality code at first.

I usually refactor my code when it is hard to understand/maintain, however, if I have to prioritize refactorings, I refactor elements that contain bugs and have smells. Since I'm already refactoring the code to get rid of the bug, I take the opportunity to remove the smell as well. The goal is to kill two birds with one stone.

I usually don't refactor buggy code when a bug is identified; I only fix it asap and leave possible refactoring for later. I apply a refactoring usually when the code is badly structured. If a bug is found in the middle of the refactoring, I will be fixed also during the refactoring process, asap.

11 - Are you aware of some issues in which code smells are harmful to the software quality? If yes could you please describe it below or put the issue link?

No
Maintainability
Maintenance
Most of the code smells are dangerous in their own ways. They usually they make the codebase hard to maintain and modify. Or they introduce subtle bugs in certain scenarios.
Efficiency, security and maintainability
God classes are the first example that comes to my mind. It doesn't have separation of concerns, do more than one job, if maintaining that code you will affect many points...there are many quality risks. It is even hard to focus the quality assurance in just one part of the application, forcing you to do a full regression.
Yes. It makes code harder to read and understand and this contributes to more bugs and more time debugging.
Coupled code, fuzzy parameters names
Yes. Security or maintenance issues
As i said before, maintainability, you can code with code smells if it is a single person codebase because you know what have you done. But when there are more developers involved in a development, code smells make then doubt about the intentions or purposes of that code smell, is it there otherwise code doesn't work?, those kind of questions make developers lose time and commit errors
Misunderstandings waiting to happen.
Code smells often indicate or lead to bigger problems. Those bigger problems can make a code base fragile, difficult to maintain, and prone to errors.
Some code smells, has a direct relationship with quality attributes such as coupling and cohesion. For instance, feature envy -> coupling and cohesion; god class -> cohesion; divergence change e shotgun surgery -> coupling.
Yes. Sometimes, when you have a method or function too long, normally this is related to centralize many tasks in only one place and that method/function is being responsible for activities that actually shouldn't be its responsibility. So, this decreases the software quality in terms of future maintenance or possible software evolution.
Sometimes it can lead to early complex architecture
Code smells use to decrease code readability.
Yes
Bad decisions of choosing patterns, the false senior developer (probably a old people in your team that you trust in a period, and when you see the disaster are made). In terms of code, inheritance, bad encapsulation, repeating yourself
Sure, and they are many. As stated in Q.6, I've struggled to read and change smelly code many times before. Too messy code is irritating, right? Once I had to prepare a project for migration across programming languages. It was a hell of a work to read some pretty large classes with dozens of lengthy and complex methods. And what can I say about the uncountable dependencies among classes that made it hard to reorganize the code every now and then?
Fault proneness, software degradation, Error proneness, maintenance difficulty
No
No

12 - Please let us know if you have any additional comments about Harmful Code.

Fix it.
If you have a Sonarqube around, and code reviews in the delivery pipeline they should be there in the first place.
Bad quality code is an effect of programmers with bad knowledge about programming. Not only programming languages, but also logic, abstractions and modeling.
Its really important detect it, even more when there are multiple teams working in the same project.
Most of the projects that I worked on got new bugs after refactorings. Maybe the concept of refactoring should be upgrade to something like: code smells + bug = refactoring.
How much harmful, much priced
It's very expensive to fix a bug that is in production, devs should analyze their solution, many times before submit their code. Testing is a very important phase of the development and the scenarios need to be real ones and complex scenarios, covering edge cases and large data input.

2 Tables: How effective are Machine Learning techniques to detect harmful code?

Algorithm	Smell	Harmful
Switch Statements		
KNeighborsClassifier	0.29	0.80
RandomForestClassifier	0.25	1.00
DecisionTreeClassifier	0.50	1.00
AdaBoostClassifier	0.33	1.00
GradientBoostingClassifier	0.25	1.00
SVM	0.50	0.89
GaussianNB	0.50	0.86
MIN	0.25	0.80
MAX	0.50	1.00

Algorithm	Smell	Harmful
Magic Number		
KNeighborsClassifier	0.50	0.73
RandomForestClassifier	0.80	1.00
DecisionTreeClassifier	0.67	0.96
AdaBoostClassifier	0.56	0.85
GradientBoostingClassifier	0.72	0.93
SVM	0.72	0.80
GaussianNB	0.25	0.68
MIN	0.25	0.68
MAX	0.80	1.00

Algorithm	Smell	Harmful
Long Identifier		
KNeighborsClassifier	0.67	0.80
RandomForestClassifier	0.75	1.00
DecisionTreeClassifier	0.75	1.00
GradientBoostingClassifier	0.75	1.00
SVM	0.67	0.80
GaussianNB	0.33	0.86
MIN	0.33	0.80
MAX	0.75	1.00

Algorithm	Smell	Harmful
Insufficient Modularization		
KNeighborsClassifier	0.64	0.90
RandomForestClassifier	0.60	1.00
DecisionTreeClassifier	0.50	0.97
AdaBoostClassifier	0.60	0.97
GradientBoostingClassifier	0.57	1.00
SVM	0.69	0.93
GaussianNB	0.67	0.62
MIN	0.50	0.62
MAX	0.69	1.00

Algorithm	Smell	Harmful
Long Parameter List		
KNeighborsClassifier	0.92	1.00
RandomForestClassifier	0.93	1.00
DecisionTreeClassifier	0.78	1.00
AdaBoostClassifier	0.93	1.00
GradientBoostingClassifier	1.00	1.00
SVM	0.92	1.00
GaussianNB	0.92	1.00
MIN	0.78	1.00
MAX	1.00	1.00

Algorithm	Smell	Harmful
Unutilized Abstraction		
KNeighborsClassifier	0.70	0.80
RandomForestClassifier	0.81	1.00
DecisionTreeClassifier	0.72	0.96
GradientBoostingClassifier	0.86	1.00
SVM	0.63	0.88
GaussianNB	0.55	0.70
MIN	0.55	0.70
MAX	0.86	1.00

Algorithm	Smell	Harmful
Cyclic-Dependent Modularization		
KNeighborsClassifier	0.50	0.74
RandomForestClassifier	0.67	0.97
DecisionTreeClassifier	0.67	0.97
AdaBoostClassifier	0.61	0.88
GradientBoostingClassifier	0.71	0.97
SVM	0.45	0.88
GaussianNB	0.52	0.58
MIN	0.45	0.58
MAX	0.71	0.97

Algorithm	Smell	Harmful
Deficient Encapsulation		
KNeighborsClassifier	0.73	0.90
RandomForestClassifier	0.65	0.98
DecisionTreeClassifier	0.61	1.00
AdaBoostClassifier	0.67	0.83
GradientBoostingClassifier	0.63	0.98
SVM	0.70	0.96
GaussianNB	0.56	0.55
MIN	0.56	0.55
MAX	0.73	1.00

Algorithm	Smell	Harmful
Long Method		
KNeighborsClassifier	0.40	0.75
RandomForestClassifier	0.75	1.00
DecisionTreeClassifier	0.29	0.86
AdaBoostClassifier	0.86	1.00
GradientBoostingClassifier	0.86	1.00
SVM	0.50	0.75
GaussianNB	0.86	1.00
MIN	0.29	0.75
MAX	0.86	1.00

Algorithm	Smell	Harmful
Long Statement		
KNeighborsClassifier	0.79	0.96
RandomForestClassifier	0.86	1.00
DecisionTreeClassifier	0.81	1.00
AdaBoostClassifier	0.64	1.00
GradientBoostingClassifier	0.86	1.00
SVM	0.75	1.00
GaussianNB	0.13	0.75
MIN	0.13	0.75
MAX	0.86	1.00

Algorithm	Smell	Harmful
Empty Catch Clause		
KNeighborsClassifier	0.86	0.86
RandomForestClassifier	0.50	1.00
DecisionTreeClassifier	0.00	0.86
AdaBoostClassifier	0.00	0.86
GradientBoostingClassifier	0.44	1.00
SVM	1.00	1.00
GaussianNB	0.00	1.00
MIN	0.00	0.86
MAX	1.00	1.00

3 Table: Which metrics are most influential on detecting harmful code?

Smell	Feature	Harmful
Cyclic-Dependent Modularization	returns	0.0699
	variables	0.0408
	unique_words_qty	0.0373
	line	0.0321
	numbers_qty	0.0213
	string_literals_qty	0.0191
	rfc	0.0170
	cbo	0.0161
	anonymous_classes_qty	0.0097
parameters	0.0064	
Deficient Encapsulation	wmc	0.0776
	parameters	0.0709
	unique_words_qty	0.0404
	returns	0.0171
	variables	0.0129
	rfc	0.0119
	line	0.0113
	number_commits	0.0016
	cbo	0.0014
total_methods	0.0011	
Empty Catch Clause	wmc	0.1467
	rfc	0.0807
	unique_words_qty	0.0371
	cbo	0.0368
	parameters	0.0281
	line	0.0267
	variables	0.0108
	returns	0.0097
	numbers_qty	0.0012
anonymous_classes_qty	0.0008	
Insufficient Modularization	wmc	0.0446
	unique_words_qty	0.0329
	rfc	0.0140
	string_literals_qty	0.0072
	variables	0.0051
	total_methods	0.0019
	total_fields	0.0012
	sub_classes_qty	0.0007
	static_methods	0.0002

Smell	Feature	Harmful
Long Identifier	math_operations_qty	0.1035
	anonymous_classes_qty	0.0516
	cbo	0.0494
	rfc	0.0251
	loc	0.0227
	numbers_qty	0.0118
	unique_words_qty	0.0074
	line	0.0058
	string_literals_qty	0.0028
	variables	0.0009
Long Method	wmc	0.0145
	unique_words_qty	0.0261
	string_literals_qty	0.0023
	rfc	0.0187
	returns	0.0257
	numbers_qty	0.0119
	number_days	0.0001
	number_commits	0.0005
	median_files	0.0002
	loc	0.0938
Long Parameter List	numbers_qty	0.1211
	parameters	0.1176
	loc	0.0671
	wmc	0.0241
	cbo	0.0146
	parenthesized_exps_qty	0.0138
	line	0.0121
	rfc	0.0073
	string_literals_qty	0.0039
	max_nested_blocks	0.0002
Long Statement	parameters	0.1951
	line	0.0321
	rfc	0.0180
	variables	0.0151
	wmc	0.0136
	unique_words_qty	0.0104
	math_operations_qty	0.0074
	returns	0.0014
	max_nested_blocks	0.0005
	numbers_qty	0.0005
Magic Number	cbo	0.0668
	returns	0.0430
	line	0.0396
	anonymous_classes_qty	0.0316
	rfc	0.0254
	parameters	0.0210
	numbers_qty	0.0188
	wmc	0.0156
	variables	0.0148
	unique_words_qty	0.0118

Smell	Feature	Harmful
Switch Statements	math_operations_qty	0.2281
	rfc	0.0000
	wmc	0.0000
	numbers_qty	0.0000
	line	0.0000
	unique_words_qty	0.0000
	number_commits	0.0000
	median_files	0.0000
	cbo	0.0000
	parameters	0.0000
Unutilized Abstraction	line	0.0723
	unique_words_qty	0.0260
	rfc	0.0243
	cbo	0.0184
	string_literals_qty	0.0179
	math_operations_qty	0.0155
	parameters	0.0088
	variables	0.0060
	wmc	0.0060
	max_nested_blocks	0.0057

4 Software Metrics (CK)

Name	Description
CBO (Coupling between objects)	Counts the number of dependencies a class has. The tools checks for any type used in the entire class (field declaration, method return types, variable declarations, etc). It ignores dependencies to Java itself (e.g. java.lang.String).
DIT (Depth Inheritance Tree)	It counts the number of "fathers" a class has. All classes have DIT at least 1 (everyone inherits java.lang.Object). In order to make it happen, classes must exist in the project (i.e. if a class depends upon X which relies in a jar/dependency file, and X depends upon other classes, DIT is counted as 2).
Number of fields	Counts the number of fields. Specific numbers for total number of fields, static, public, private, protected, default, final, and synchronized fields.
Number of methods	Counts the number of methods. Specific numbers for total number of methods, static, public, abstract, private, protected, default, final, and synchronized methods.
NOSI (Number of static invocations)	Counts the number of invocations to static methods. It can only count the ones that can be resolved by the JDIT.
RFC (Response for a Class)	Counts the number of unique method invocations in a class. As invocations are resolved via static analysis, this implementation fails when a method has overloads with same number of parameters, but different types.
WMC (Weight Method Class) or McCabe's complexity	It counts the number of branch instructions in a class.
LOC (Lines of code)	It counts the lines of count, ignoring empty lines.
LCOM (Lack of Cohesion of Methods)	Calculates LCOM metric. This is the very first version of metric, which is not reliable. LCOM-HS can be better (hopefully, you will send us a pull request).
Quantity of returns	The number of return instructions.
Quantity of loops	The number of loops (i.e., for, while, do while, enhanced for).
Quantity of comparisons	The number of comparisons (i.e., ==).
Quantity of try/catches	The number of try/catches.
Quantity of parenthesized expressions	The number of expressions inside parenthesis.
String literals	The number of string literals (e.g., "John Doe"). Repeated strings count as many times as they appear.
Quantity of Number	The number of numbers (i.e., int, long, double, float) literals.
Quantity of Math Operations	The number of math operations (times, divide, remainder, plus, minus, left shift, right shift).
Quantity of Variables	Number of declared variables.
Max nested blocks	The highest number of blocks nested together.
Quantity of Anonymous classes, subclasses, and lambda expressions	The Quantity of Anonymous classes, subclasses, and lambda expressions.
Number of unique words	Number of unique words in the source code. See WordCounter class for details on the implementation.
Usage of each variable	How much each variable was used inside each method.
Usage of each field	How much each field was used inside each method.

5 Developer Metrics

Name	Type	Tool Used	Description
Number of Commits (NC)	Developers' Experience	PyDriller	this metric represents the number of commits authored by a developer;
Number of Active Days in Project (NADP)	Developers' Experience	PyDriller	this metric indicates how many days a developer has been active, i.e., committing;
Number of Days in Project (NDP)	Developers' Experience	PyDriller	this metric counts the number of days that a developer has been associated to a project, independently if he is contributing or not;
Number of issues Activities (NIA)	Developers' Experience	GitHub API	this metric measures the number of issues opened or closed by a developer;
Number of Pull Requests Activities (NPRA)	Developers' Experience	GitHub API	this metric measures the number of pull requests opened or closed by a developer;
Number of Tests Included (TI)	Technical Contribution Norms	PyDriller	this metric measures the quantity of commits that contain tests. To extract it, we adopted the procedure defined by [18]. First, we retrieve all the files modified in a commit authored by a developer. Then, we check how many files contain the "test" word in its pathname;
Median of Modified Files (MMF)	Technical Contribution Norms	PyDriller	this metric measures the median of modified files among all the commits authored by a developer;
Median of Lines Changed (MLC)	Technical Contribution Norms	PyDriller	this metric represents the median of changed lines among all the commits authored by a developer. A changed line can be an addition or a deletion in a commit;
Number of Followers (NF)	General Community Status	WebCrawler	this metric represents the number of followers that a developer has on GitHub;
Number of Public Repository (NPR)	General Community Status	WebCrawler	this metric counts the number of public repositories owned by a developer on GitHub;
Number of Public Gists (NPG)	General Community Status	WebCrawler	this metric represents the number of public Gists 2 owned by a developer. A Gist is a tool designed to share single files, parts of source code, or full applications created by a developer. Such tool may be very important to stimulate the reuse of software artifacts.