iscte

INSTITUTO
UNIVERSITÁRIO
DE LISBOA

Streamlining Code Smells:
Using Collective Intelligence and Visualization

José Vicente Pereira dos Reis

PhD in Information Science and Technology

Supervisors:
Doctor Fernando Brito e Abreu, Associate Professor,
Iscte– Instituto Universitário de Lisboa

Doctor Glauco de Figueiredo Carneiro, Assistant Professor,
Federal University of Sergipe (UFS), Brazil

June, 2022

TECNOLOGIAS
E ARQUITETURA

Department of Information Science and Technology

Streamlining Code Smells:
Using Collective Intelligence and Visualization

José Vicente Pereira dos Reis

PhD in Information Science and Technology

Jury:

Doctor Fernando Manuel Marques Batista,Associate Professor
Iscte– Instituto Universitário de Lisboa (President)
Doctor Silvia Mara Abrahão, Associate Professor
Universitat Politècnica de València
Doctor Marcelo de Almeida Maia, Full Professor
Universidade Federal de Uberlândia
Doctor Luís Miguel Martins Nunes, Associate Professor
Iscte– Instituto Universitário de Lisboa
Doctor Fernando Brito e Abreu, Associate Professor
Iscte– Instituto Universitário de Lisboa

June, 2022

Streamlining Code Smells:
Using Collective Intelligence and Visualization

Copyright © 2022, José Vicente Pereira dos Reis, School of Technology and Architecture, University Institute of Lisbon.
The School of Technology and Architecture and the University Institute of Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

[This page has been intentionally left blank]

To my wife and daughter.

[This page has been intentionally left blank]

Acknowledgements

When I decided to do this Ph.D., I expected a long, complicated, and arduous journey, with ups and downs, but also enriching and exciting. However, several people have contributed to mitigating the difficulties arising and to make this Ph.D. even more fascinating, for which I would like to thank them.

First of all, I would like to thank my supervisor Professor Fernando Brito e Abreu for his unconditional support, advice, suggestions, and criticism during these years. He started as an advisor and ended as a friend. I hope it has been as rewarding for Professor Fernando Brito e Abreu to orient me as it has been for me to be oriented by him. My deepest thanks.

I would also like to thank my co-supervisor, Professor Glauco Carneiro, for his advice, support, criticism, and suggestions. It was a great pleasure to have had his collaboration, help, and friendship in realizing this Ph.D.

I would like to thank the member of my follow-up committee, Professor Luís Nunes, for his comments, suggestions, and our conversations. Thank you very much.

Being sure that I will forget someone and that many other people also deserve my thanks because they contributed directly or indirectly to this thesis, I want to thank in a very special way :

My colleague and friend, Professor Vítor Basto-Fernandes, this thesis would not be possible without him because his involvement in the crowd experiments was fundamental to making it possible. I have deep gratitude for him.

My friends and colleagues in this long journey, Américo Rio and João Caldeira, for their contributions, readiness to help, discussions, criticisms, proposals for improvement, and many other things. Their support was tireless in the realization of this thesis.

Professor Craig Anslow, from Victoria University of Wellington, New Zealand, for his collaboration in realizing the SLR, suggestions, and improvements. Without him, we would not have had many answers to the surveys.

My many thanks to the technology students who collaborated in the experimental study and without whom the study would not exist.

Last but not least, I would like to thank my parents and brothers, for all their love, support, and encouragement and for always believing in me, especially my father, who died during the course of this thesis.

And finally, to the most important people in my life, my wife and daughter, for their patience, support, and understanding. For the moments when I was not present in their lives due to this thesis.

Thank you.

Lisboa, June of 2022
José Vicente Pereira dos Reis

Abstract

Context. Code smells are seen as major source of technical debt and, as such, should be detected and removed. Code smells have long been catalogued with corresponding mitigating solutions called refactoring operations. However, while the latter are supported in current IDEs (e.g., Eclipse), code smells detection scaffolding has still many limitations. Researchers argue that the subjectiveness of the code smells detection process is a major hindrance to mitigate the problem of smells-infected code.

Objective. This thesis presents a new approach to code smells detection that we have called CrowdSmelling and the results of a validation experiment for this approach. The latter is based on supervised machine learning techniques, where the wisdom of the crowd (of software developers) is used to collectively calibrate code smells detection algorithms, thereby lessening the subjectivity issue.

Method. In the context of three consecutive years of a Software Engineering course, a total "crowd" of around a hundred teams, with an average of three members each, classified the presence of 3 code smells (Long Method, God Class, and Feature Envy) in Java source code. These classifications were the basis of the oracles used for training six machine learning algorithms. Over one hundred models were generated and evaluated to determine which machine learning algorithms had the best performance in detecting each of the aforementioned code smells.

Results. Good performances were obtained for God Class detection (ROC=0.896 for Naive Bayes) and Long Method detection (ROC=0.870 for AdaBoostM1), but much lower for Feature Envy (ROC=0.570 for Random Forrest).

Conclusions. Obtained results suggest that Crowdsmelling is a feasible approach for the detection of code smells, but further validation experiments are required to cover more code smells and to increase external validity.

Keywords: crowdsmelling, code smells, crowdsourcing, code smells detection, machine learning, software quality

[This page has been intentionally left blank]

Resumo

Contexto. Os cheiros de código são a principal causa de dívida técnica (technical debt), como tal, devem ser detectados e removidos. Os cheiros de código já foram há muito tempo catalogados juntamente com as correspondentes soluções mitigadoras chamadas operações de refabricação (refactoring). No entanto, embora estas últimas sejam suportadas nas IDEs actuais (por exemplo, Eclipse), a deteção de cheiros de código têm ainda muitas limitações. Os investigadores argumentam que a subjectividade do processo de deteção de cheiros de código é um dos principais obstáculo à mitigação do problema da qualidade do código.

Objectivo. Esta tese apresenta uma nova abordagem à detecção de cheiros de código, a que chamámos CrowdSmelling, e os resultados de uma experiência de validação para esta abordagem. A nossa abordagem de CrowdSmelling baseia-se em técnicas de aprendizagem automática supervisionada, onde a sabedoria da multidão (dos programadores de software) é utilizada para calibrar colectivamente algoritmos de detecção de cheiros de código, diminuindo assim a questão da subjectividade.

Método. Em três anos consecutivos, no âmbito da Unidade Curricular de Engenharia de Software, uma "multidão", num total de cerca de uma centena de equipas, com uma média de três membros cada, classificou a presença de 3 cheiros de código (Long Method, God Class, and Feature Envy) em código fonte Java. Estas classificações foram a base dos oráculos utilizados para o treino de seis algoritmos de aprendizagem automática. Mais de cem modelos foram gerados e avaliados para determinar quais os algoritmos de aprendizagem de máquinas com melhor desempenho na detecção de cada um dos cheiros de código acima mencionados.

Resultados. Foram obtidos bons desempenhos na detecção do God Class (ROC=0,896 para Naive Bayes) e na detecção do Long Method (ROC=0,870 para AdaBoostM1), mas muito mais baixos para Feature Envy (ROC=0,570 para Random Forrest).

Conclusões. Os resultados obtidos sugerem que o Crowdsmelling é uma abordagem viável para a detecção de cheiros de código, mas são necessárias mais experiências de validação para cobrir mais cheiros de código e para aumentar a validade externa.

Palavras-chave: crowdsmelling, cheiros de código, crowdsourcing, deteção de cheiros de código, aprendizagem automática, qualidade do software.

[This page has been intentionally left blank]

Contents

Acknowledgements ix

Abstract xi

Resumo xiii

List of Figures xix

List of Tables xxi

Listings xxiii

Acronyms xxv

I Fundamentals 1

1 Introduction 3
- 1.1 Motivation and Scope 4
  - 1.1.1 Code Smells and Its Relevance on Software Design 4
  - 1.1.2 Code Smell Detection and visualization 4
  - 1.1.3 Collective intelligence 7
- 1.2 Research Drivers 8
  - 1.2.1 Research Problems 8
  - 1.2.2 Research Questions 9
  - 1.2.3 Main Contributions 9
- 1.3 Dissertation Outline 11
- 1.4 Summary 13

2 State of the Art 15
- 2.1 Introduction 17
- 2.2 Related work 17
- 2.3 Research Methodology 20
  - 2.3.1 Planning the Review 20
  - 2.3.2 Conducting the Review 21
- 2.4 Results and Analysis 25
  - 2.4.1 Overview of studies 26

2.4.2 Approach for CS detection (**F1**) . . . 28
2.4.3 Dataset availability (**F2**) . . . 31
2.4.4 Programming language (**F3**) . . . 31
2.4.5 Code smells detected (**F4**) . . . 33
2.4.6 Machine Learning techniques used (**F5**) . . . 34
2.4.7 Evaluation of techniques (**F6**) . . . 35
2.4.8 Detection tools (**F7**) . . . 36
2.4.9 Thresholds definition (**F8**) . . . 37
2.4.10 Validation of techniques (**F9**) . . . 38
2.4.11 Replication of the studies (**F10**) . . . 40
2.4.12 Visualization techniques (**F11**) . . . 41
2.5 Discussion . . . 43
2.5.1 Research Questions (**RQ**) . . . 43
2.5.2 SLR validation . . . 45
2.5.3 Validity threats . . . 46
2.6 Conclusion . . . 48
2.6.1 Conclusions on this SLR . . . 48
2.6.2 Open issues . . . 49
2.7 Summary . . . 49

II CS Detection and Visualization **51**

3 *Crowdsmelling*: The use of collective knowledge in CS detection **53**
3.1 Introduction . . . 55
3.2 Related Work . . . 55
3.2.1 Crowd and collaborative-based approaches . . . 55
3.2.2 Multiple ML models based approaches . . . 57
3.3 Experiment Planning . . . 58
3.3.1 Research Questions . . . 58
3.3.2 Participants . . . 58
3.3.3 Data . . . 59
3.3.4 CS . . . 61
3.3.5 Code Metrics . . . 61
3.3.6 Machine Learning Techniques Experimented . . . 61
3.3.7 Model Evaluation . . . 62
3.3.8 Process . . . 63
3.4 Results . . . 67
3.4.1 **RQ1. What is the performance of ML techniques when trained with data from the crowd?** . . . 67
3.4.2 **RQ2. What is the best ML model to detect each one of the three CS?** . . . 70
3.4.3 **RQ3. Is it possible to use Collective Knowledge for CS detection?** . . . 72
3.5 Discussion . . . 74

3.5.1 Research Questions (RQ) 74
3.5.2 Implications and limitations of the *Crowdsmelling* Approach 76
3.5.3 Threats to validity 77
3.6 Summary 79

4 Code Smells Visualization **81**
4.1 Introduction 82
4.2 Visualization Survey 82
4.2.1 Survey and Samples 83
4.2.2 Survey Results 83
4.3 *Smelly Maps* as SourceMiner Views 89
4.4 Summary 91

III Crowdsmelling: a ML-based crowdsourcing approach for code smells detection **93**

5 Crowdsmelling Tool **95**
5.1 Introduction 96
5.2 Motivation 96
5.3 Related work 96
5.3.1 Code smells detection tools 97
5.3.2 ML-based code smells detection 98
5.4 Crowdsmelling 100
5.4.1 Proposed approach 100
5.4.2 Proposed architecture for an application using approach 101
5.4.3 Application usage scenarios 102
5.5 Summary 106

IV Conclusion **109**

6 Conclusion and Future Work **111**
6.1 Introduction 112
6.2 Thesis Synthesis 112
6.3 Main Contributions 113
6.4 Research Opportunities 114

Bibliography **117**

Appendices **131**

A Systematic Literature Review Materials **131**
A.1 Studies included in the review 132
A.2 Studies after applying inclusion and exclusion criteria (phase 3) 139

A.3 Quality assessment . . . 148
A.4 Description of code smells detected in the studies, according to the authors . . 151
A.5 Frequencies of code smells detected in the studies . . . 156

B Crowdsmelling Materials **159**
B.1 Code metrics . . . 160

C Architectures of the crowdsmelling tool versions **163**
C.1 Version 1 - Eclipse plugin and Azure Machine Learning . . . 164
C.1.1 Eclipse IDE plugin . . . 164
C.1.2 Machine Learning Component . . . 164
C.2 Version 2 - Eclipse plugin and Weka . . . 164
C.3 Version 3 - Microservices Architecture . . . 166

D Eclipse Java Metamodel **167**
D.1 Eclipse Java Metamodel . . . 168

List of Figures

2.1 Stages of the study selection process 24
2.2 Selected studies per research question (RQ) 25
2.3 Trend of publication years 26
2.4 Type of publication venue 26
2.5 Programming languages and number of studies that use them 32
2.6 Number of languages used in each study 32
2.7 Number of code smells detected by number of studies 34
2.8 Summary of main findings 42
2.9 Relations between findings and research questions 43

3.1 Process of CS classification by the developer 63
3.2 Process of creation of the datasets and evaluation of the ML techniques by the researcher 65
3.3 Process of testing the variance between ML models 66

4.1 Answers to the question: "The vast majority of code smells detection studies do not propose visualization features for their detection" 84
4.2 Answers to the question: "The vast majority of existing code smells visualization studies did not present evidence of its usage upon large software systems" 85
4.3 Answers to the question: "Software visualization researchers have not adopted specific visualization related taxonomies to support the identification of code smells" 86
4.4 Answers to the question: "If visualization related taxonomies were used in the implementation of code smells detection tools, that could enhance their effectiveness." 87
4.5 Answers to the question: "Which of the following visual attributes have you implemented in tools targeting the support of code smells identification?" 88
4.6 Answers to the question: "The combined use of collaboration (among software developers) and visual resources may increase the effectiveness of code smells detection." 89
4.7 An extended reference model for MVIEs (from [23]) 90

5.1 Component Diagram 101
5.2 Use Case Diagram 103
5.3 The CrowdSmelling approach process 104
5.4 The Code smells detection process 105
5.5 The code smells classification process 105

C.1 Crowdsmelling Eclipse IDE plugin 164
C.2 Feature Envy training workflow 165
C.3 Feature Envy predictive workflow 165

D.1 Eclipse Java Metamodel - Java Project Structure 168
D.2 Eclipse Java Metamodel - Type Components 169
D.3 Eclipse Java Metamodel - Abstract Syntax Tree Components 170

List of Tables

1.1 Code smells described in Martin Fowler's Catalog [43] . . . 5
1.2 Smell taxonomy . . . 6
1.3 Correspondence between chapters of the thesis and papers . . . 12

2.1 Inclusion criteria . . . 22
2.2 Exclusion criteria . . . 22
2.3 Interpretation of the Kappa results . . . 23
2.4 Quality criteria (Stage 4) . . . 23
2.5 Number of studies by score obtained after application of the quality assessment criteria (stage 4) . . . 24
2.6 Top-ten cited papers, according to Google Scholar . . . 27
2.7 CS detection approaches used . . . 29
2.8 Top ten open-source software projects used in the studies . . . 31
2.9 Code smells detected in more than 3 studies . . . 33
2.10 ML algorithms used in the studies . . . 35
2.11 Metrics used to evaluate the detection techniques . . . 36
2.12 Number of studies that developed a tool and its approach . . . 37
2.13 Number of studies that use thresholds in CS detection . . . 38
2.14 Tools / approach used by the studies for validation . . . 39
2.15 Summary of survey results . . . 47

3.1 Teams whose CS detection was included in the oracles . . . 59
3.2 Datasets (Oracles) and their composition . . . 60
3.3 *Long Method*: *ROC* Area results for the ML algorithms trained by the 3 years datasets 67
3.4 *God Class*: *ROC* Area results for the ML algorithms trained by the 3 years datasets 68
3.5 *Feature Envy*: *ROC* Area results for the ML algorithms trained by the 3 years datasets 69
3.6 *Long Method*: Performance of the code smell prediction models . . . 71
3.7 *God Class*: Performance of the code smell prediction models . . . 72
3.8 *Feature Envy*: Performance of the code smell prediction models . . . 73

4.1 Different scopes of code smells . . . 90

[This page has been intentionally left blank]

Listings

Acronyms

ANOVA one-way ANalysis Of VAriance.
API Application Program Interface.
AST Abstract Syntax Tree.

BBN Bayesian Belief Networks.
BPMN Business Process Model.

CS Code Smells.
CSV Comma Separated Values.

EJM Eclipse Java Model.
EJMM Eclipse Java Metamodel.

GA Genetic Algorithms.

IDE Integrated Development Environment.

JDT Eclipse Java Development Tools.

M2DM MetaModel Driven Measurement.
ML Machine Learning.
MSA MicroServices Architecture.
MVIE Multiple Views Interactive Environments.

OCL Object Constraint Language.
OO Object-Oriented.

REST Representational State Transfer.
ROC Receiver Operating Characteristic.

SLR Systematic Literature Review.

SM Systematic Mapping Study.

SO Service-Oriented.

Part I.

Fundamentals

PART I: FUNDAMENTALS

Introduction
Chapter 1

State of the Art
Chapter 2

PART II: CODE SMELLS DETECTION AND VISUALIZATION

Crowdsmelling: The use of collective knowledge in code smells detection
Chapter 3

Smelly Maps
Chapter 4

PART III: CROWDSMELLING: A ML-BASED CROWDSOURCING APPROACH FOR CODE SMELLS DETECTION

Crowdsmelling Tool
Chapter 5

PART IV: CONCLUSION

Conclusion
Chapter 6

This part covers the motivation, scope, research problems and main contributions of this work and highlights the fundamental topics, such as: code smells, code smells detection and visualization. It also presents a Systematic Literature Review about code smells.

CHAPTER 1.

Introduction

Contents

1.1 Motivation and Scope 4
1.1.1 Code Smells and Its Relevance on Software Design 4
1.1.2 Code Smell Detection and visualization 4
1.1.3 Collective intelligence 7
1.2 Research Drivers 8
1.2.1 Research Problems 8
1.2.2 Research Questions 9
1.2.3 Main Contributions 9
1.3 Dissertation Outline 11
1.4 Summary 13

This chapter introduces the motivation and scope, describes the problems faced in detecting and visualizing code smells, and suggests methods to overcome them. Finally, it summarizes the contributions made to the detection and visualization of code smells.

1.1 Motivation and Scope

In this chapter, we present the motivation and scope that led to the realization of this thesis. We begin in section 1.1.1 by introducing the problem of software maintenance, the influence that code smells have on it, and the principal code smells. Then, in section 1.1.2, we focus on the problems of detecting and visualizing code smells.

1.1.1 Code Smells and Its Relevance on Software Design

Software maintenance has historically been the Achilles' heel of the software life cycle [2]. Maintenance tasks are incremental modifications to a software system that aim to add or adjust some functionality or correct some design flaws and fix some bugs. It has been found that feature addition, modification, bug fixing, and design improvement can cost as much as 80% of total software development cost [131]. In addition, it is shown that software maintainers spend around 60% of their time in understanding code [148]. Therefore, as much as almost half (80%x60%=48%) of total development cost may be spent on understanding code. This high cost can be reduced by the availability of tools to increase code understandability, adaptability, and extensibility [78].

In software development and maintenance, especially in complex systems, the existence of Code Smells (CS) jeopardizes the quality of the software and hinders several operations such as code reuse. Code smells are not bugs since they do not prevent a program from functioning, but rather symptoms of software maintainability problems [144]. However, they often correspond to the violation of fundamental design principles and may slow down software evolution (e.g., due to code misunderstanding) or increase the risk of bugs or failures in the future. Code smells can then compromise software quality in the long term by inducing technical debt [9].

In this context, the detection of CS or anti-patterns (undesirable patterns, said to be recipes for disaster [16]) is a topic of special interest since it prevents code misunderstanding and mitigates potential maintenance difficulties.According to the authors of [122], there is a subtle difference between a CS and an anti-pattern: the former is a kind of warning for the presence of the latter. This thesis will not explore that difference, thereby only referring to the CS concept.

The most relevant CS are cataloged, and the most widely used catalog was compiled by Martin Fowler [43] and describes 22 CS. Table 1.1 presents the 22 CSs of Fowler's catalog and their description according to [142]. Other researchers, such as Van Emden and Moonen [34], have subsequently proposed more CS and provided the first formalization of code smells. Mäntylä et al. [79] and Wake [135] proposed two initial taxonomies for code smells (see Table 1.2). In recent years, CSs have been cataloged for other object-oriented programming languages, such as Matlab [46], Python [27], and Java Android-specific CS [61, 98], which confirms the increasing recognition of their importance.

1.1.2 Code Smell Detection and visualization

Many techniques and tools have been proposed in the literature for detecting and visualizing code smells [114], but the former faces a few challenges. The first is that code smells lack a formal definition [136]. Therefore, their detection is highly subjective (e.g., dependent on the

Table 1.1: Code smells described in Martin Fowler's Catalog [43]

Code Smell	Description
Alternative Classes with Different Interfaces	Classes that mostly do the same things, but have methods with different signatures
Data Class	Classes with fields and getters and setters not implementing any function in particular
Data Clumps	Clumps of data items that are always found together weather within classes or between classes
Divergent Change	One class is commonly changed in different ways for different reasons
Duplicated Code	Same or similar code structure repeated within a class or between classes
Feature Envy	A method that seems more interested in another class other than the one it's actually in. Fowler recommends putting a method in the class that contains most of the data the method needs
God (Large) Class	Class takes too many responsibilities relative to the classes with which it is coupled. The God Class centralizes the system functionality in one class, which contradicts the decomposition design principles
Long Method	The methods is very large compared to the other methods in the same class. Long Method centralizes the class functionality in one method
Inappropriate Intimacy	Two classes are overly intertwined
Incomplete library class	Libraries lacking on specific functionality
Lazy Class	A class with not enough functionality
Long Parameter List	Provide a method with just enough data so that it can obtain everything it needs
Message Chains	This is the case in which a client has to use one object to get another, and then use that one to get to another, etc. Any change to the intermediate relationships causes the client to have to change.
Middle Man	A class is delegating almost everything to another class
Misplaced Class	In large packages it happens often that a class needs the classes from other packages more than those from its own package.
Parallel Inheritance Hierarchies	Each is required to make a subclass of one class, is required also to make a subclass of another
Primitive Obsession	Use primitives data types instead of small objects for simple tasks (such as phone numbers, money, etc.)
Refused Bequest	Subclasses do not want or need everything they inherit
Shotgun Surgery	A change in a class results in the need to make a lot of little changes in several classes
Speculative Generality	Over-generalized code in an attempt to predict future needs
Switch Statements	Conditionals depending of type leading to duplication
Temporary Field	Consists of fields that are used as temporary variables. This means that a value assigned to such a field is not used by any method except for the method containing the assignment

Table 1.2: Smell taxonomy

Class	**Code Smells**
Bloaters	Long Method, Large Class, Primitive Obsession, Long Parameter List, Data Clumps
Object-Orientation Abusers	Switch Statements, Temporary Field, Refused Bequest, Alternative Classes with Different Interfaces, Parallel Inheritance Hierarchies
Change Preventers	Divergent Change and Shotgun Surgery
Dispensables	Lazy Class, Data Class, Duplicate Code, Speculative Generality
Encapsulators	Message Chains, Middle Man
Couplers	Feature Envy, Inappropriate Intimacy
Others	Incomplete Library Class, Comments

developer's experience). Second, due to the dramatic growth in the size and complexity of software systems in the last four decades [54], it is not feasible to detect code smells thoroughly without tools.

A factor that exacerbates the complexity of CS detection is that practitioners have to reason at different abstraction levels: some CS are found at the class level, others at the method level, and even others encompass both method and class levels simultaneously (e.g., *Feature Envy*). This means that once a CS is detected, its extension/impact must be conveyed to the developer to allow him to take appropriate action (e.g., a refactoring operation). For instance, the representation of a *Long Method* (circumvented to a single method) will be rather different from that of a *Shotgun Surgery* that can spread across a myriad of classes and methods. Therefore, besides the availability of appropriate CS detectors, we need suggestive and customized CS visualization features to help practitioners understand their manifestation. Nevertheless, there are only a few primary studies aimed at CS visualization.

We classify CS visualization techniques in two categories: (i) the detection is done through a non-visual approach, the visualization being performed to show CS location in the code itself, (ii) the detection is performed through a visual approach. In the Systematic Literature Review (SLR) that we present in section 2.1, we address these two categories.

Most of the proposed CS visualization techniques show them inside the code itself. For certain systems, this technique works, but it is too meticulous for a global refactoring strategy, especially when working with large legacy systems. Therefore, a more macro approach is needed to present CS in a more aggregated form without losing information. Unfortunately, there are few primary studies in that direction.

Manual CS detection requires code inspection and human judgment and is therefore unfeasible for large software systems. Furthermore, CS detection is influenced (and hampered) by the subjectivity of their definition, as reported by Mantyla et al. [80], based on the results of experimental studies. For example, they observed that the degree of agreement in the evaluation of CS was high for the simplest CS, but when developers evaluated more complex CS

such as *Lazy Class* and *Middle Man*, the degree of agreement was lower. The main reason reported for this was the lack of knowledge about more complex CS because the latter naturally require a better understanding of the code. In other words, they suggested that experience may mitigate the subjectivity issue, and indeed they observed that experienced developers reported more complex CS than novices did. However, they also concluded that the CS' commonsense detection rules expressed in natural language could also cause misinterpretation.

Automated CS detection, mainly in object-oriented systems, involves the use of source code analysis techniques, often metrics-based [72]. Unfortunately, despite research efforts dedicated to this topic in recent years, the availability of automatic detection tools for practitioners is still scarce, especially when compared to the number of existing detection methods (see section 2.4.7).

Many researchers proposed CS detection techniques. However, most studies are only targeted to a small range of existent CS, namely *God Class*, *Long Method* and *Feature Envy*. Moreover, only a few studies are related with the application of calibration techniques in CS detection (see section 2.4.5 and 2.4.7). The latter relates to defining a set of parameters for an algorithm that results in creating a predictive model.

Considering the diversity of existing techniques for CS detection, it is important to group the different approaches into categories for a better understanding of the type of technique used. Kessentini et al. [62] classified those approaches into seven categories: metric-based approaches, search-based approaches, symptom-based approaches, visualization-based approaches, probabilistic approaches, cooperative-based approaches, and manual approaches. The most popular code smells detection approach is metric-based. The latter is based on the application of detection rules that compare the values of relevant metrics extracted from the source code with empirically identified thresholds. However, these techniques present some problems, such as subjective interpretation, a low agreement between detectors [40], and threshold dependability.

To overcome the aforementioned limitations of code smell detection, researchers recently applied supervised machine learning techniques that can learn from previous datasets without needing any threshold definition [6, 31]. However, the main impediment for applying those techniques is the scarcity of publicly available oracles, i.e., tagged datasets for training detection algorithms.

1.1.3 Collective intelligence

The use of collective intelligence in problem-solving is not a new phenomenon. It has been used in various sciences such as biology, social sciences, engineering, computer science, etc., for many years and refers to intelligence in groups [73].

In the book *The Wisdom of Crowds*, James Surowiecki [129] presents a set of experiments and events where a group of people can, under certain conditions, achieve better results than the most intelligent individual in the group. Surowiecki argues that groups are remarkably intelligent and do not need to be dominated by exceptionally intelligent people in order to make intelligent decisions. However, to get good results from a group, it must meet four conditions that characterize wise crowds: i) diversity of opinion (each person should have some private information, even if it's just an eccentric interpretation of the known facts), ii) independence

(people's opinions are not determined by the opinions of those around them), iii) decentralization (people are able to specialize and draw on local knowledge), and iv) aggregation (some mechanism exists for turning private judgments into a collective decision).

Leimeistern, in his article *Collective Intelligence* [73], presents a set of potential areas of application of Collective Intelligence and refers to Collective Intelligence as: "Decomposing collective intelligence etymologically, the term *collective* describes a group of individuals who are not required to have the same attitudes or viewpoints. Different members can reveal different perspectives and approaches, and thus leading to better explanations or solutions to a given problem. *Intelligence* refers to the ability to learn, to understand, and to adapt to an environment by using own knowledge."

1.2 Research Drivers

The starting point of our research process was the formulation of the main problems we are undertaking (section 1.2.1). This led us to the research questions presented in the section 1.2.2. To answer these questions, we had to use a methodology and conduct experiments that supported our conclusions about the benefits and effectiveness of our proposals. The main contributions of this dissertation are described in section 1.2.3.

1.2.1 Research Problems

The goal of our research work is to address the following problems in the area of code smells detection and visualization:

- **RP1. Code smells detection.** Although in recent years we have seen an evolution in the automatic detection of smells (see the related work in section 2.1), that kind of support is still very limited, namely when compared with their mitigating solutions, known as refactoring operations [41]. Without automated support, code smells detection becomes a fastidious manual process, often taken as an unfeasible overhead when there is (there is always) pressure to deliver a new version. The main reasons why the current state of the art in the automatic detection of code smells is still poor are: i) most code smells have subjective (natural language-based) definitions [132, 136], and ii) automation requires a different detection algorithm for each code smell, and that implies training data to guarantee its *Accuracy*. Looking more carefully, we see that the first cause subsumes the second: subjectivity hampers the availability of appropriate training data.

- **RP2. Code smells visualization.** When dealing with large, complex software systems, code smells awareness features are particularly important since their distribution and collateral effects may spread a lot. Effective, yet non-intrusive, visualization features should allow to a) spot the location of code smells, b) diagnose their cause, and c) warn of their potential hazardous consequences. Several software visualization metaphors were proposed in the literature [32]. A comprehensive one considers a system as a city [139], packages as neighborhoods, classes as buildings, and methods as their floors. In such a setup, it would not be upfront to represent, for instance, code smells that occur within

a hierarchy, simply because class hierarchies are not clearly mapped into this appealing city metaphor. This counterexample highlights the need to carry out further research to figure out which metaphor is the most appropriate for each smell.

The systematic literature review results in section 2.1 confirm that these research problems are still open issues. Also, surveys of researchers in the areas of code smells detection and software visualization reinforce the importance and relevance of these problems.

1.2.2 Research Questions

To address the research problems of section 1.2.1, the following research questions were formulated:

- **RQ1**: Is it possible to use Collective Knowledge for code smells detection?
- **RQ2**: What is the performance of machine learning techniques when trained with data from the crowd and hypothetically more realistic?
- **RQ3**: How can we visualize CS location in large software systems?

Each RQ is related to the corresponding RP in section 1.2.1. Thus, RQ1 and RQ2 are related to RP1, and RQ3 is related to RP2. RQ2 presumes that RQ1 has a positive answer.

1.2.3 Main Contributions

This section presents a summary of the main contributions achieved with the development of this dissertation:

1. ***Crowdsmelling*** approach for Code smells detection.

   We propose the concept of ***crowdsmelling*** – use of collective intelligence in the detection of code smells – to mitigate the aforementioned problems of subjectivity and lack of calibration data required to obtain accurate detection model parameters. *Crowdsmelling* is a collaborative crowdsourcing approach based on machine learning, where the wisdom of the crowd (of software developers) is used to collectively calibrate code smells detection algorithms (one per each code smell type). The applications based in collective intelligence, where the contribution of several users allows attaining benefits of scale and/or other types of competitive advantage, are gaining increasing importance in Software Engineering [127] and other areas [11, 12]. Some of the most notable examples of crowdsourcing in Software Engineering are *crowdtesting* [120], *code snippets recommendation* [105] or *stack overflow* [126].

2. ML-based crowdsourcing approach for code smells detection.

   We propose a crowdsourcing approach based on supervised Machine Learning (ML) algorithms to mitigate the subjectivity problem to detect code smells. We implemented the approach in a tool dubbed *CrowdSmelling Checker* and presented its usage scenarios to detect code smells.

The front-end of the proposed tool is a plugin installed in each developer's Integrated Development Environment (IDE) that computes metrics from the source code and sends them to a MicroServices Architecture (MSA), requesting the location of detected code smells. That architecture includes a scientific workflow management system, an execution platform for ML algorithms, and a database management system. All services communicate through Representational State Transfer (REST) interfaces.

In brief, *CrowdSmelling Checker* includes a frontend embedded as a plugin in the IDE and a backend in the cloud. The frontend captures explanatory variables from the currently opened software project in the IDE and sends their values to the backend. In the backend, a set of Machine Learning (ML) algorithms suggest the locations of code smells occurrences and feds them back to the frontend. The software developer will either accept (true positives) or reject the proposed code smells occurrences (false positives) or suggest the location of non-detected occurrences (false negatives). Crowdsourcing is fundamental in this approach since the feedback received from the crowd of software developers on false negatives, and true and false positives are used to train multiple ML algorithms, allowing the dynamic calibration and choice of the best alternative for the detection of code smells. As such, each developer will participate in the collective enrichment of the training sets used for calibrating the ML algorithms. The corresponding ML algorithm will be retrained whenever a training set size increases by a given delta. This progressive calibration will improve the detection process. The input factors (aka explanatory variables) to the aforementioned ML-based detection algorithms will be formally expressed in OCL [137] the upon MetaModel Driven Measurement (M2DM) approach [1], upon the Eclipse Java Metamodel (EJMM) defined in [55]. The latter was obtained by reverse engineering two Eclipse Java Development Tools (JDT) components: the Eclipse Java Model (EJM) and the Eclipse Abstract Syntax Tree (AST). According to the M2DM approach, the explanatory variables will be formalized in Object Constraint Language (OCL) [137] upon the EJMM.

3. Code smells visualization

   We also propose code smells visualization at different abstraction levels, aiming at increasing software quality awareness and facilitating refactoring decisions upon large software systems.

   Visualization provides perceivable cues to several aspects of the data under analysis to reveal patterns and behaviors that would otherwise remain "under the radar" [125]. Code smells are defined at different scopes (within one method, within one class, within a class hierarchy, across several methods, across several classes). A visualization feature for a code smell of the first kind (within one method) seems straightforward: the best way is adding some type of flag, usually in the code editor's margin, in the place where it was found. Even though classes may be large (i.e., spread across several screen heights), it is still acceptable to use the flagging technique for representing the location of code smells of the second kind (within one class), for instance, close to the class header. As for the other three kinds of code smells, we need to identify adequate visualization mechanisms

since they may spread across many methods or classes. We will call *Smelly Maps* to these views, and they will act as a front-end for the more complex code smells, facilitating the comprehension of their side-effects and the diagnosis of their cure.

Based on a well-known reference model for information visualization from Card et al. [20], Carneiro and Mendonça extended and adapted it to the context of *SourceMiner*, a Multiple Views Interactive Environments (MVIE) implemented as an Eclipse plugin [21, 22], that allows visualizing software attributes at different levels of abstraction (packages, types, and operations). We offer *Smelly Maps* as a set of new views in *SourceMiner*, which will work in cooperation with the *CrowdSmelling* tool.

4. List of Publications

   The following papers were produced for this thesis, which contains the main contents presented in this dissertation:

   4.1 J. Pereira dos Reis, F. Brito e Abreu, and G. de Figueiredo Carneiro, "Code Smells Incidence: Does It Depend on the Application Domain?," in QUATIC, 2016, pp. 172–177. IEEE. https://doi.org/10.1109/QUATIC.2016.044

   4.2 J. Pereira dos Reis, F. Brito e Abreu, and G. de Figueiredo Carneiro,"Code smells detection 2.0:Crowdsmelling and visualization." in 2017 12th Iberian Conference on Information Systems and Technologies (CISTI). June 2017, pp. 1–4. IEEE. https://doi.org/10.23919/CISTI.2017.7975961

   4.3 J Caldeira, FB e Abreu, JP dos Reis, J Cardoso, "Assessing Software Development Teams' Efficiency using Process Mining."in 2019 International Conference on Process Mining (ICPM), 2019, pp. 65-72. IEEE. https://doi.org/10.1109/ICPM.2019.00020

   4.4 J. Pereira dos Reis, F. Brito e Abreu, G. de Figueiredo Carneiro, and C. Anslow. "Code Smells Detection and Visualization: A Systematic Literature Review." in Archives of Computational Methods in Engineering 29, 47-94 (2022). Springer. https://doi.org/10.1007/s11831-021-09566-x

   4.5 J. Pereira dos Reis, F. Brito e Abreu, and G. de Figueiredo Carneiro, "Crowdsmelling: A preliminary study on using collective knowledge in code smells detection."In: Empiricial Software Engineering 27(3), 69 (2022). Springer. https://doi.org/10.1007/s10664-021-10110-5

   4.6 J Caldeira, FB e Abreu, J Cardoso, JP dos Reis, "Unveiling process insights from refactoring practices."in Computer Standards & Interfaces 81, C (2021). Elsevier. https://doi.org/10.1016/j.csi.2021.103587

   4.7 J. Pereira dos Reis, F. Brito e Abreu, and G. de Figueiredo Carneiro, "Crowdsmelling: a ML-based crowdsourcing approach for code smells detection." - Submitted.

1.3 Dissertation Outline

This thesis is based on a set of articles, where each chapter corresponds to an article or part of an article (except for chapter 6-Conclusion and Future Work), as presented in table 1.3.

Table 1.3: Correspondence between chapters of the thesis and papers

Chapter	Paper name
1-Introduction	Code smells detection 2.0:Crowdsmelling and visualization
2-State of the Art	Code Smells Detection and Visualization: A Systematic Literature Review
3-Crowdsmelling: The use of collective knowledge in code smells detection	Crowdsmelling: A preliminary study on using collective knowledge in code smells detection
4-Code Smells Visualization	Code Smells Detection and Visualization: A Systematic Literature Review
5-Crowdsmelling Tool	Crowdsmelling: a ML-based crowdsourcing approach for code smells detection (Submitted)
6-Conclusion and Future Work	Several papers

The option to publish all chapters was intended to validate the research and obtain feedback from reviewers, thus improving the work done.

This thesis is organized into four parts, as shown in Figure 1.1, where each part contains a set of chapters, six in total. Each of these chapters is briefly described as follows:

Part I - Fundamentals . Introduces this dissertation, the fundamental topics, and an SLR.

- **Chapter 1 - Introduction.** Provides context to this work, identifies some of the main problems in research and details the solutions prescribed to mitigate them. Finally, it summarizes the benefits and highlights the dissertation structure.

- **Chapter 2 - State of the Art.** It gives an overview of the related work, proposes a taxonomy to categorize it, and identifies research gaps within the code smells detection and visualization area.

Part II - Code Smells Detection and Visualization. The solution proposed and its applications.

- **Chapter 3 - Crowdsmelling: The use of collective knowledge in code smells detection.** Presents the results of a validation experiment for the *Crowdsmelling* approach, where the wisdom of the crowd (of software developers) is used to collectively calibrate code smells detection algorithms, thereby lessening the subjectivity issue.

- **Chapter 4 - *Smelly Maps.*** *Smelly Maps*, to visualize code smells.

Part III - Crowdsmelling: a ML-based crowdsourcing approach for code smells detection. Implements the *Crowdsmelling* approach in a tool.

- **Chapter 5 - Crowdsmelling Tool.** Presents the implementation of the *Crowdsmelling* approach in a tool and presents its usage scenarios to detect code smells.

Part IV - Conclusion. Draws the conclusions and raises new research opportunities.

- **Chapter 6- Conclusions and Future Work.** Concludes and summarizes the achievements of this dissertation and discusses future work.

1.4 Summary

This first chapter aims to provide an overview of the research work carried out in this thesis. We begin with an introduction to what code smells are and their relevance in the software design section 1.1.1, followed by an approach to code smell detection and visualization, presenting the various types of approaches and main problems in detection and visualization, section 1.1.2. Next, research problems, research questions and main contributions expected from the thesis are described, respectively, in sections 1.2.1, 1.2.2 and 1.2.3. Finally, section 1.3 provides an outline with a brief explanation of what is included in each chapter.

[This page has been intentionally left blank]

Chapter 2

State of the Art

Contents

2.1 Introduction 17
2.2 Related work 17
2.3 Research Methodology 20
2.3.1 Planning the Review 20
2.3.2 Conducting the Review 21
2.4 Results and Analysis 25
2.4.1 Overview of studies 26
2.4.2 Approach for CS detection (**F1**) 28
2.4.3 Dataset availability (**F2**) 31
2.4.4 Programming language (**F3**) 31
2.4.5 Code smells detected (**F4**) 33
2.4.6 Machine Learning techniques used (**F5**) 34
2.4.7 Evaluation of techniques (**F6**) 35
2.4.8 Detection tools (**F7**) 36
2.4.9 Thresholds definition (**F8**) 37
2.4.10 Validation of techniques (**F9**) 38
2.4.11 Replication of the studies (**F10**) 40
2.4.12 Visualization techniques (**F11**) 41
2.5 Discussion 43
2.5.1 Research Questions (**RQ**) 43
2.5.2 SLR validation 45
2.5.3 Validity threats 46
2.6 Conclusion 48
2.6.1 Conclusions on this SLR 48
2.6.2 Open issues 49

2.7 Summary 49

This chapter present the protocol design, execution and findings of a Systematic Literature Review (SLR) on Code Smells Detection and Visualization.

2.1 Introduction

The purpose of this chapter is to systematically review the published research on code smells detection and visualization. We chose to perform an SLR, as this provides a fair evaluation of the current state of the art in the area, using a trustworthy, rigorous, and auditable methodology.

Summing up, the main objectives for this review are:

- What are the main techniques for the detection of CS and their respective effectiveness reported in the literature?
- What are the visual approaches and techniques reported in the literature to represent CS and therefore support practitioners to identify their manifestation?

2.2 Related work

We will present the related work in chronological order.

Zhang et al. [146] presented a systematic review on CS, where more than 300 papers published from 2000 to 2009 in the main journals from IEEE, ACM, Springer and other publishers were investigated. After applying the selection criteria, the 39 most relevant ones were analyzed in detail. The authors revealed that *Duplicated Code* is the most widely studied CS. The authors' results indicate that most studies focus on the development of methods and tools for the automatic detection of CS, with few studies reporting the impact that CS have and therefore a phenomenon that was far from being fully understood.

A vast literature review was conducted by Rattan et al. [110] to study software clones (aka *Duplicate Code*) in general and software clone detection in particular. The research was focused on a systematic set of 213 papers published in 11 leading journals and 37 premier conferences and workshops out of a total of 2039 articles. An empirical assessment of clone detection tools/techniques is provided. Clone management, its benefits, and cross cutting nature is reported. Studies involving nine different types of clones are presented, as well as thirteen intermediate representations and 24 match detection techniques. In conclusion, the authors call for a better understanding of the possible advantages of software clone management and recognize the need for semantic and model clone detection techniques to be developed.

Rasool and Arshad [109] presented a review on various detection tools and techniques for mining CS. They used the classification presented by Kessentini et al. [62] to classify the CS detection techniques presented in the various papers and compared the various approaches based on their key characteristics. They also compared the results of the techniques and tools presented in the review studies, and presented a critical analysis, where the limitations for the different tools were identified. The authors concluded, for example, that there was still no consensual definition of CS definitions in the research community, and that there was a lack of systems to serve as a reference standard for the evaluation of existing techniques.

Al Dallal [3] performed an SLR that identifies and analyzed techniques that identify opportunities for refactoring object-oriented code. A total of 47 studies were analyzed, and some of the most important conclusions reached by the authors were: the most frequent refactoring activities were Extract Method, Move Method, and Extract Class; the most used approach in

identifying refactoring opportunities was quality metrics-oriented; open-source systems were the most used to validate researchers' results, the most used being JHotDraw, and the dominant programming language in datasets was Java.

Fernandes et al. [37] presented the findings of a SLR on CS detection tools. They found in the literature a reference to 84 tools, but only 29 of them were available online for download. These 84 tools used at least six different detection techniques to detect a total of 61 CS. The dominant programming languages in tool development and for CS detection were Java, C, and C++. In a second phase of the SLR, the authors presented a comparative study of four detection tools concerning two CS: *Long Method* and *Large Class*. The findings showed that the results obtained by the tools were redundant for the same CS. Finally, this SLR concluded that the three CS most detected by the tools were the *Long Method*, *Large Class*, and *Duplicated Code*.

Singh and Kaur [122] published a SLR on refactoring with respect to CS. Although the title appears to focus on refactoring, different types of techniques for identifying CS and antipatterns are discussed in depth. The authors claimed that this work was an extension of the one published in [3]. They found 1053 papers in the first stage, which were reduced to 325 just based on the paper title. Then, based on the abstract, they trimmed down that number to 267. Finally, a set of 238 papers was selected after applying inclusion and exclusion criteria. This SLR includes primary studies from the early ages of digital libraries till September 2015. Some conclusions regarding detection approaches were that 28.15% of researchers applied automated detection approaches to discover the CS, while empirical studies were used by a total of 26.89% of researchers . The authors also pointed out that Apache Xerces, JFreeChart and ArgoUML were among the most targeted systems. They also reckon that *God Class* and *Feature Envy* are the most recurrently detected CS.

Gupta et al. [49] performed a SLR based on publications from 1999 to 2016 and 60 papers, screened out of 854, were deeply analyzed. The objectives of this SLR were to provide an overview of the investigation carried out in the area of CS, identify the techniques used in the detection and find out which CS were the most detected with the various detection approaches. The authors of this SLR concluded that the *Duplicate Code* was the most used CS in papers, the impact of CS was poorly studied, the majority of papers focused on the study of detection techniques and tools, and a significant inverse correlation between detection techniques and CS has been performed on the basis of CS. They also identified four CS from Fowler's catalog, whose detection was not reported in the literature: *Inappropriate Intimacy*, *Primitive Obsession*, *Comments* and *Incomplete Library Class*.

Alkharabsheh et al. [4] performed a Systematic Mapping Study (SM) where they analyzed studies published between 2000 and 2017, in a total of 395 articles, published in journals, proceedings, and book chapters, in the area of Design Smells Detection. The main conclusions of this SLR pointed at to the need of standardizing the concepts, requiring greater collaboration among international research teams there was no correlation between detection techniques and efficiency of results, and all automatic detection tools produced a binary detection result (having the CS or not); the non-existence of a CS corpus common to several detection tools was also a significant problem; it was complicated to compare the results of different techniques due to the absence of benchmarks, and the homogeneity in the performance of the indicators was low; another important finding was the detection of CS positively influenced the quality of

the code.

Santos et al. [118] investigated the effect of CS on software development, the *CS effect*. They reached three main results: the need to study more the effects of CS in software development because there was still a lack of understanding of these effects; in the systematic review of the 67 papers, there was no strong evidence of the correlation between CS and the effort to maintain the software; due to the low agreement on the detection of CS, manual evaluation of CS should not be trusted. Finally, the authors made suggestions to improve knowledge of the effects of CS: improving the factors that influence the human perception of CS, e.g., practitioners' experience; perform a better classification of CS according to their relevant characteristics.

Sabir et al. [116] Aimed to identify the similarities and differences in the identification of CS in Object-Oriented (OO) and Service-Oriented (SO) Software. Thus, in the SLR, they investigated the main techniques used to detect CS in different paradigms of Software Engineering from OO to SO. They performed a SLR based on publications from January 2000 to December 2017 and selected 78 papers. The authors concluded that: the most used CS in the literature were *Feature Envy*, *God Class*, *Blob*, and *Data Class*; on the opposite side were CS as the *Yo-Yo Problem*, *Intensive coupling*, *Unnamed Coupling*, and *Interface Bloat*, less mentioned in the literature. Mainly two techniques in the detection of smells were used in the literature: static source code analysis and dynamic source code analysis based on dynamic threshold adaptation, e.g., using a genetic algorithm, instead of fixed thresholds for smells detection.

The SLR proposed by Azeem et al. [7] investigated the usage of Machine Learning approaches in the field of CS between 2000 and 2017. Out of the 2456 papers initially obtained, only 15 used ML approaches. The study of the 15 papers was conducted from four different perspectives: (i) CS considered, (ii) setup of ML approaches, (iii) design of the evaluation strategies, and (iv) a meta-analysis on the performance achieved by the models proposed so far. The authors concluded that: the most used CS in the literature are *God Class*, *Long Method*, *Functional Decomposition*, and *Spaghetti Code*; The most widely used ML algorithms in CS detection are Decision Trees and Support Vector Machines; there are several questions that the research community did not yet answered and that should be focused on in the future, e.g., tools to capture the severity of CS; finally, they argue that ML techniques in CS detection can still be improved.

Kaur [57] examined 74 primary studies focused on CS detection through search-based approaches. The results obtained indicated that there was no benchmark system to compare the results of the different detection approaches, making it difficult to validate the results. The authors carried out an experiment with 2 tools that used a search-based approach (*jDeodorant* and *CodeNose*) to compare the detection results. The problems of determining threshold values in metric-based detection techniques were still unknown and unreliable, as well as the detection of CS in large systems. Other findings state that most tools are not publicly available; few authors evaluated their approaches in commercial systems, using essentially open-source systems; Java was the most used programming language, and the most used CS were *Feature Envy*, *Long Method*, *Duplicate Code*, and *Long Parameter List*.

Sobrinho et al. [124] conducted an extensive Systematic Literature Review to determine the state of the art on bad smells with 351 works produced between 1990 and 2017. The SLR was structured through five Thematic Areas, the 5 W's, having reached the following main

conclusions: (i) Bad smell types (Which). The most studied code smell was the Duplicate Code, present in 69.8% of the studies, followed by Large Class, Feature envy, Long Method, and Data Class; The co-occurrence of Large Class with Long Method was also detected in 41 papers. (ii) Interest on smells over time (When). There has been an oscillation in the number of papers published since 1990; however, interest in this research topic has increased over the years, with new researchers appearing every year. (iii) Aims, findings, and settings (What). The two main aims of the papers are to study the detection and impact of code smells on software maintenance. Regarding the impact, the authors identified that there are sometimes contradictions among the studies. These contradictions seem to be due to the wide range of tools and systems used in the experimental settings, suggesting a lack of well-designed benchmarks. (iv) Researchers (Who). The authors have different levels of interest in code smells, and few of them address studies with a wide range of smells. It was also detected an influence in the publications and code smells studied due to scientific connections among researchers. (v) Distribution of papers among venues (Where). Some venues, like ICSE (International Conference on Software Engineering), SCAM (Working Conference on Source Code Analysis and Manipulation), ICSME (International Conference on Software Maintenance and Evolution), WCRE (Working Conference on Reverse Engineering), ICPC (International Conference on Program Comprehension), CSMR (Conference on Software Maintenance and Reengineering), and EMSE (Empirical Software Engineering), have a higher proportion of Duplicated Code papers. However, Since 2004, the interest in other bad smells has increased, where TSE (IEEE Transactions on Software Engineering) and ICSME are the venues with the highest proportion of studies.

The scope and coverage of the current SLR goes beyond those of aforementioned SLRs, mainly because it also covers the CS visualization aspects. The latter is important to show programmers the scope of detected CSs, to help deciding whether to refactor or not. A good visualization becomes even more important if one takes into account the subjectivity existing in the definition of CS, which leads to the detection of many false positives.

2.3 Research Methodology

An SLR consists of a sequence of specific and rigorous methodological steps for the purpose of reviewing research literature, reducing bias and enhancing replication [15] and [68]. SLRs rely on well-defined and evaluated review protocols to extract, analyze, and document results, as the stages conveyed in Figure 2.1. This section describes the methodology applied for the phases of planning, conducting and reporting the review.

2.3.1 Planning the Review

Identify the needs for a systematic review. Search for evidences in the literature regarding the main techniques for CS detection and visualization, in terms of (i) strategies to detect CS, (ii) effectiveness of CS detection techniques, (iii) approaches and techniques for CS visualization.

Research Questions. We aim to answer the following questions, by conducting a methodological review of published research results:

RQ1. *Which techniques have been reported in the literature for the detection of CS?* The list of the main techniques reported in the literature for the detection of CS can provide a comprehensive view for both practitioners and researchers, supporting them in selecting a technique that best fits their daily activities, as well as highlighting which of them deserve more effort to be analyzed in future experimental studies.
RQ2. *What literature has reported on the effectiveness of techniques aiming at detecting CS?* The goal is to compare the techniques among themselves, using parameters such as *Accuracy*, *Precision* and *Recall*, as well as the classification of automatic, semi-automatic or manual approaches.
RQ3. *What are the techniques and resources used to visualize CS and therefore support the practitioners to identify CS occurrences?* The visualization of CS occurrences is a key issue for its adoption in the industry, due to the variety of CS, their scope (e.g. within methods or classes, or among methods or classes).

These three research questions are somehow related to each other. In fact, any detection algorithm after being implemented, should be tested and evaluated to verify its effectiveness, which causes RQ1 and RQ2 to be closely related. RQ3 encompasses two possible situations: i) CS detection is done through visual techniques, and ii) the latter are only used for representing CS previously detected with other techniques; therefore, there is also a close relationship between RQ1 and RQ3.
Publications Time Frame. We conducted a SLR in journals, conferences papers and book chapters from January 2000 to June 2019.

2.3.2 Conducting the Review

This phase is responsible for executing the review protocol.
Identification of research. Based on the research questions, keywords were extracted and used to search the primary study sources. The search string is presented as follows and used the same strategy cited in [26]:

("code smell"OR "bad smell") AND (visualization OR visual OR representation OR identification OR detection) AND (methodology OR approach OR technique OR tool)

Selection of primary studies. The following steps guided the selection of primary studies.

Stage 1 - Search string results automatically obtained from scientific repositories - ACM Digital Library, IEEE Xplore, ISI Web of Science, Science Direct, Scopus and Springer Link were selected based on their relevance as sources in Software Engineering [145]. The search was conducted using the specific syntax of each database, considering only the title, keywords, and abstract. The search was configured in each repository to select only papers carried out within the defined period. The automatic search was complemented by a backward snowballing manual search, following the guidelines of Wohlin [140]. Duplicates were discarded.

Stage 2 - Analyse titles, abstracts and keywords to identify potentially relevant studies - Studies that were clearly irrelevant to the search were discarded at this stage. If there was any doubt about whether a study should be included or not, it was included for consideration on a later stage.

Stage 3 - Apply inclusion and exclusion criteria on reading the introduction, methods and conclusion sections - Selected studies in previous stages were reviewed, by reading the introduction, methodology and conclusion sections. Afterwards, inclusion and exclusion criteria were applied (see Table 2.1 and Table 2.2). At this stage, in case of doubt preventing a conclusion, the study was read in its entirety.

Table 2.1: Inclusion criteria

Criterion	Description
IC1	The publication venue should be a "journal" or "conference proceedings" or "book".
IC2	The primary study should be written in English.
IC3	The primary work is an empirical study or have "lessons learned"(experience report).
IC4	If several papers report the same study, the latest one will be included.
IC5	The primary work addresses at least one of the research questions.

Table 2.2: Exclusion criteria

Criterion	Description
EC1	Studies not focused on code smells.
EC2	Short paper (less than 2000 words, excluding numbers) or unavailable in full text.
EC3	Secondary and tertiary studies, editorials/prefaces, readers' letters, panels, and poster-based short papers.
EC4	Works published outside the selected time frame.
EC5	Code Smells detected in non-object oriented programming languages.

The reliability of the inclusion and exclusion criteria of a publication in the SLR was assessed by applying Fleiss' Kappa [38]. Fleiss' Kappa is a statistical measure for assessing the reliability of agreement between a fixed number of raters when classifying items. We used the Kappa statistic [84] to measure the level of agreement between the researchers. Kappa result is based on the number of answers with the same result for all observers [71]. Its maximum value is 1, when the researchers have almost perfect agreement, and it tends to -1 when there is no agreement between them. Table 2.3 shows the interpretation of this coefficient according to Landis & Koch [71].

We asked two seniors researchers to classify, individually, a sample of 31 publications to analyze the degree of agreement in the selection process through the Fleiss' Kappa [38]. The selected sample was the set of the most recent publications (last 2 years) from phase 2. The result of the degree of agreement showed a substantial (good) level of agreement between the two researchers (Kappa = 0.631).

The 102 studies resulting from this phase are listed in Appendix A.2.

Stage 4 - Obtain primary studies and make a critical assessment of them - A list of primary studies was obtained and later subjected to critical examination using the 8 quality criteria set

Table 2.3: Interpretation of the Kappa results

Kappa values	Degree of agreement
<0.00	Poor
0.00 - 0.20	Slight
0.21 - 0.40	Fair(Weak)
0.41 - 0.60	Moderate
0.61 - 0.80	Substantial (Good)
0.81 - 1.00	Almost perfect (Very Good)

out in Table 2.4. Some of these quality criteria were adapted from those proposed by Dyba and Dingsøyr [33]. In the QC1 criterion we evaluated venue quality based on its presence in the CORE rankings portal[1]. In the QC4 criterion, the relevance of the study to the community was evaluated based on the citations present in Google Scholar[2] using the method of Belikov and Belikov [10]. The grading of each of the 8 criteria was done on a dichotomous scale ("YES"=1 or "NO"=0). For each primary study, its quality score was determined by summing up the scores of the answers to all the 8 questions. A given paper satisfies the Quality Assessment criteria if reaches a rating equal or higher to 4. Among the 102 papers resulting from stage 3, 19 studies [11, 16, 19, 22, 32, 36, 57, 71, 73, 77, 82, 83, 85, 86, 87, 90, 91, 95, 102] (see Appendix A.2) were excluded because they did not reach the minimum score of 4 (Table 2.5), while 83 passed the Quality Assessment criteria. All 83 selected studies are listed in Appendix A.1 and the details of the application of the quality assessment criteria are presented in Appendix A.3.

Table 2.4: Quality criteria (Stage 4)

Criterion	Description
QC1	Is the venue recognized in CORE rankings portal?
QC2	Was the data collected in a way that addressed the research issue?
QC3	Is there a clear statement of findings?
QC4	Is the relevance for research or practice recognized by the community?
QC5	Is there an adequate description of the validation strategy?
QC6	The study contains the required elements to allow replication?
QC7	The evaluation strategies and metrics used are explicitly reported?
QC8	Is a CS visualization technique clearly defined?

Data extraction. All relevant information on each study was recorded on a spreadsheet. This information was helpful to summarize the data and map it to its source. The following data were extracted from the studies: (i) title and authors; (ii) year; (iii) type of article (journal, conference, book chapter); (iv) name of conference, journal or book; (v) number of Google Scholar citations at the time of evaluation; (vi) answers to research questions; (vii) answers to

[1] http://www.core.edu.au/
[2] https://scholar.google.com/

Table 2.5: Number of studies by score obtained after application of the quality assessment criteria (stage 4)

Resulting score	Number of studies	% studies
1	3	2.9%
2	4	3.9%
3	12	11.8%
4	15	14.7%
5	30	29.4%
6	32	31.4%
7	6	5.9%
8	0	0.0%

quality criteria.

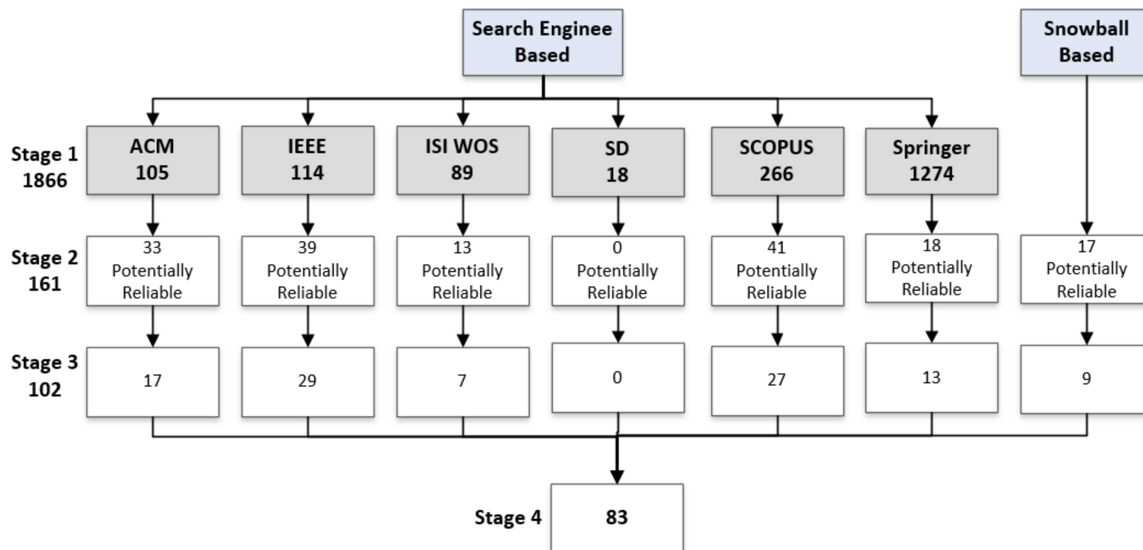


Figure 2.1: Stages of the study selection process

Data Synthesis. This synthesis is aimed at grouping findings from the studies in order to: identify the main concepts covered regarding CS detection and visualization, conduct a comparative analysis on the characteristics of the study, type of method adopted, and identify how the three research questions (*RQ1*, *RQ2* and *RQ3*) were addressed on each study. Other information was synthesized when necessary. For the data synthesis process, we have used the meta-ethnography method [91] as a guide.

Conducting the Review. We started the review with an automatic search followed by a manual search, to identify potentially relevant studies and later applied the inclusion/exclusion criteria and finally the quality criteria. In some search engines we had to adapt the search string, but always keeping its primary meaning and scope. The manual search consisted in studies published in conference proceedings, journals and books, that were selected by the authors through backward snowballing in primary studies. These studies were equally analyzed and

Figure 2.1 presents them as 17 studies. We tabulated everything on a spreadsheet so as to facilitate the subsequent phase of identifying potentially relevant studies. Figure 2.1 presents the results obtained from each electronic database used in the search, which resulted in 1866 articles considering all databases.

Potentially Relevant Studies. The results obtained from both the automatic and manual search were included on a single spreadsheet. Papers with identical title, author(s), year and abstract were discarded as redundant. At this stage, we registered an overall of 1866 articles *(Stage 1)*. We then read titles and abstracts to identify relevant studies resulting in 161 papers *(Stage 2)* including the corresponding to the backward snowballing procedure. At *Stage 3* we read introduction, methodology and conclusion in each study and then we applied the inclusion and exclusion criteria, resulting in 102 papers. In *Stage 4*, after applying the quality criteria *(QC)* the remaining 83 papers were analysed to answer the three research questions - RQ1, RQ2 and RQ3.

2.4 Results and Analysis

In this section we present the results and the analysis of this SLR, which will allow us to answer the 3 research questions (RQ1, RQ2 and RQ3), based on the quality criteria and findings (F). Figure 2.2 shows the relationship between the selected studies and the research questions they answer. In the figure we can see that 72 studies discuss RQ1 issues, 61 papers address RQ2 issues, and 17 studies address issues related to RQ3. All selected studies are listed in Appendix A.1 and referenced as *"S"* followed by the number of the paper.

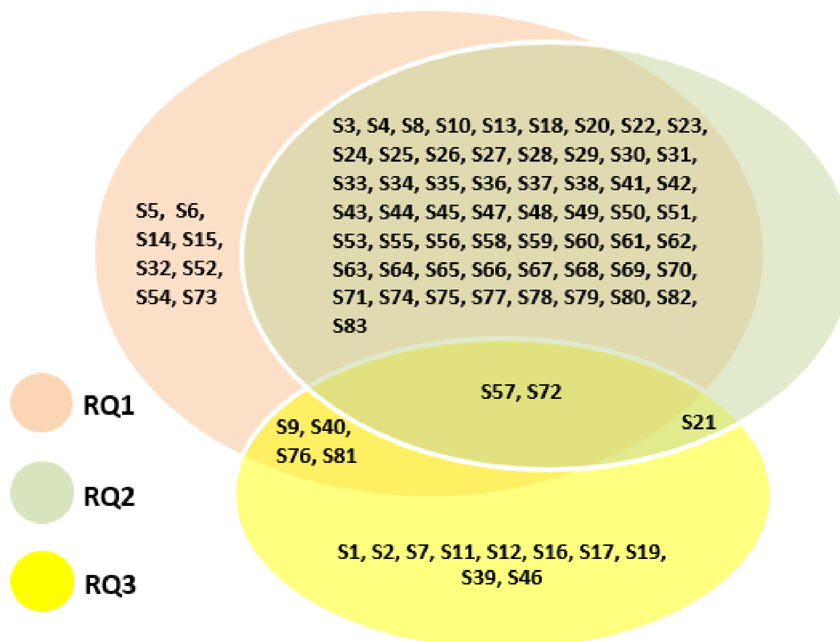


Figure 2.2: Selected studies per research question (RQ)

2.4.1 Overview of studies

The study selection process (Figure 2.1) resulted in 83 studies selected for data extraction and analysis. Figure 2.3 presents the temporal distribution of primary studies. Note that 78.3% primary studies have been published after 2009 (last 10 years) and that 2016 and 2018 were the years that had the largest number of studies published. We can conclude that although research in Software Engineering started several decades ago (since the 1970s), research on CS detection is much more recent, with most papers published in the last decade.

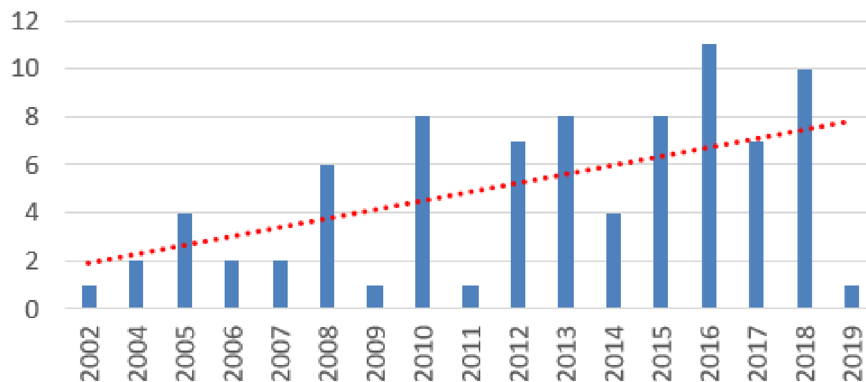


Figure 2.3: Trend of publication years

In relation to the type of publication venue (Figure 2.4), the majority of the studies were published in conference proceedings 76%, followed by journals with 23%, and 1% in books.

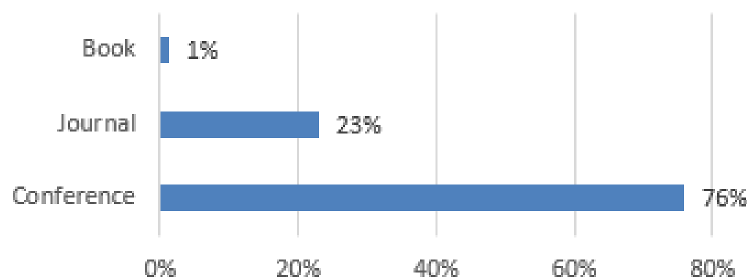


Figure 2.4: Type of publication venue

According to Google Scholar citations in September 2019[3], the top ten studies regarding their scientific impact included in the SLR are shown in Table 2.6. As can be verified by their respective citation numbers, these studies are indicative of the importance of the issues presented in this SLR and the impact these studies have on the literature. Table 2.6 also shows a summary of the distribution of the most important studies according to the corresponding RQ. In the following items, we present a summary of the studies, in descending order of the number of citations.

A brief review of each of the top cited paper follows:

[S9] - RQ1 and RQ3 are addressed in this paper that got the highest number of citations. It introduces a systematic way of detecting CS by defining detection strategies based in four steps:

[3]Data obtained in 22/09/2019

Table 2.6: Top-ten cited papers, according to Google Scholar

Studies	Cited by	Research Question
S9	964	RQ1 and RQ3
S26	577	RQ1 and RQ2
S3	562	RQ1 and RQ2
S1	423	RQ3
S10	245	RQ1 and RQ2
S4	240	RQ1 and RQ2
S7	184	RQ3
S21	174	RQ1 and RQ2
S45	157	RQ1 and RQ2
S15	156	RQ1 and RQ3

Step 1: Identify Symptoms; Step 2: Select Metrics; Step 3: Select Filters; Step 4: Compose the Detection Strategy. It describes how to explore quality metrics, set thresholds for these metrics, and create a set of rules to identify CS. Finally, visualization techniques are used to present the detection result, based in several metaphors.

[S26] - The DECOR method for specifying and detecting code and design smells is introduced. Using a consistent vocabulary and domain-specific language to automatically create detection algorithms, this approach enables defining smells at a high level of abstraction. Four design smells are identified by DECOR, namely *Blob*, *Functional Decomposition*, *Spaghetti Code*, and *Swiss Army Knife*, and the algorithms are evaluated in terms of *Precision* and *Recall*. This study addresses RQ1 and RQ2, and is one of the most used studies for validation / comparison of results in terms of *Accuracy* and *Recall* of detection algorithms.

[S3] - Issues related to RQ1 and RQ2 are discussed. This paper proposes a mechanism called "detection strategies" for producing a metric-based rules approach to detect CS with detection strategies, implemented in the IPLASMA tool. This method captures deviations from good design principles and consists of defining a list of rules based on metrics and their thresholds for detecting CS.

[S1] - A visualization approach supported by the jCOSMO tool, a CS browser that performs fully automatic detection and visualizes smells in Java source code, is proposed. This study focuses its attention on two CS, related to Java programming language, i.e., *instanceof* and *typecast*. This paper discusses issues related to RQ3.

[S10] - This paper addresses RQ1 and RQ2 and proposes the Java Anomaly Detector (JADET) tool for detecting object usage anomalies in programs. To conclude properties that are almost always fulfilled, JADET uses conceptual analysis, and it lists the failures as anomalies. This approach is based on identifying usage patterns.

[S4] - This paper presents a metric-based heuristic detection approach capable of detecting two CS, namely *Lazy Class* and *Temporary Field*. For a systematic description of CS, the authors propose a model consisting of 3 parts: a CS name, a description of its characteristics, and

heuristics for its detection. An empirical study is also reported, to justify the choice of metrics and thresholds for detecting CS. This paper discusses issues related to RQ1 and RQ2.

[S7] - This paper addresses RQ3 and presents a visualization framework for quality analysis and understanding of large-scale software systems. Programs are represented using metrics. The authors consider that fully automatic approaches are more efficient, but there is no control in context, while manual analysis is inaccurate and time-consuming. As such, they present a semi-automatic approach, which they claim is a good compromise between the two.

[S21] - This paper proposes an approach based on Bayesian Belief Networks (BBN) to identify and detect CS in applications. This approach implements the detection rules defined in DECOR [S26] and manages the uncertainty of the detection through the BBN's. The detection results are presented in the form of a probability that a class has a defect type. This paper discusses issues related to RQ1 and RQ2.

[S45] - This paper addresses RQ1 and RQ2, and proposes an approach called HIST (Historical Information for Smell deTection) to detect five different CS (*Divergent Change*, *Shotgun Surgery*, *Parallel Inheritance*, *Feature Envy*, and *Blob*). HIST explores change history information mined from versioning systems to detect CS, by analyzing co-changes among source code artifacts over time.

[S15] - This last paper in the top ten most cited ones addresses RQ1 and RQ3. It presents an Eclipse plug-in (JDeodorant) that automatically identifies Type-Checking CS in Java source code, and allows their elimination by applying appropriate refactorings. JDeodorant is one of the most used tools for validation / comparison of results in terms of *Accuracy* and *Recall* of detection algorithms.

2.4.2 Approach for CS detection (F1)

The first finding to be analyzed is the approach applied to detect CS, that is, the steps required to accomplish the detection process. For example, in the metric-based approach [72], we need to know the set of source code metrics and corresponding thresholds for the target CS.

Considering the diversity of existing techniques for CS detection, it is important to group the different approaches into categories for a better understanding of the type of technique used. Thus, we will classify the existing approaches for CS detection into seven (7) broad categories, according to the classification presented by Kessentini et al. [62]: metric-based approaches, search-based approaches, symptom-based approaches, visualization-based approaches, probabilistic approaches, cooperative-based approaches and manual approaches.

Classifying studies in one of the seven categories is not an easy task because some studies use intermediate techniques for their final technique. For example, several studies classified as symptom-based approaches use symptoms to describe CS, although detection is performed through a metric-based approach.

Table 2.7 shows the classification of the studies in the seven broad categories. The most used approaches are search-based, metric-based, and symptom-based, being used in 30.1%, 24.1% and 19.3% of the studies, respectively. The least used approaches are the cooperative-based and the manual ones, each being used in only one of the selected studies.

Table 2.7: CS detection approaches used

Approaches	Nº of studies	% Studies	Studies
Search-Based	25	30.1%	S5, S6, S14, S22, S24, S28, S33, S34, S36, S37, S42, S43, S45, S48, S51, S53, S55, S56, S71, S74, S75, S77, S78, S79, S83
Metric-Based	20	24.1%	S3, S9, S29, S38, S44, S47, S49, S52, S58, S60, S63, S64, S66, S67, S68, S70, S72, S73, S81, S82
Symptom-based	16	19.3%	S4, S8, S13, S15, S20, S23, S26, S30, S31, S32, S52, S57, S59, S61, S62, S69
Visualization-based	12	14.5%	S1, S2, S7, S11, S12, S16, S17, S19, S21, S39, S46, S76
Probabilistic	10	12.0%	S10, S18, S25, S27, S35, S40, S50, S54, S65, S80
Cooperative-based	1	1.2%	S41
Manual	1	1.2%	S52

2.4.2.1 Search-based approaches

Search-based approaches are influenced by contributions in the domain of Search-Based Software Engineering (SBSE). SBSE uses search-based methods to solve problems of Software Engineering optimization. Most techniques in this category apply ML algorithms. The main advantage of ML-based approaches is that they are automatic, so they do not require users to have extensive knowledge of the area. However, the success of these techniques depends on the quality of data sets to allow training ML algorithms.

2.4.2.2 Metric-based approaches

The metric-based approach is one of the the most commonly used. The use of quality metrics to improve the quality of software systems is not a new idea and, for more than a decade, metric-based CS detection techniques have been proposed. This approach consists in creating a rule, based on a set of metrics and respective thresholds, to detect each CS.

The main problem with this approach is that there is no consensus on the definition of CS. As such, there is no consensus on the standard threshold values for the detection of CS. Finding the best fit threshold values for each metric is complicated because it requires a significant calibration effort [62]. Threshold values are one of the main causes of the discrepancy in the results of different techniques.

2.4.2.3 Symptom-based approaches

This approach is based on describing CS symptoms, such as class roles and structures, which are then translated into detection algorithms to identify CS. Kessentini et al. [62] defines two main limitations to this approach:

- there exists no consensus in defining symptoms;

- due to the high number of existing CSs, describing their symptoms manually, producing rules and translating these into detection algorithms can be a very hard work; as a consequence, symptoms-based approaches are considered as slow and inaccurate.

Other authors [109] add more limitations, such as the definition of appropriate threshold values, when converting symptoms into detection rules, requiring a great effort of analysis. The *Accuracy* of these approaches is poor due to the various interpretations of the same symptoms.

2.4.2.4 Visualization-based approaches

Visualization-based techniques usually consist of a semi-automated process to support developers in the identification of CS. The data visually represented to this end is mainly enriched with metrics (metric-based approach) throughout specific visual metaphors.

This approach has the advantage of using visual metaphors, which reduces the complexity of dealing with a large amount of data. The disadvantages are those inherent to human intervention: (i) they require great human expertise, (ii) time-consuming, (iii) human effort, and (iv) error-prone. Thus, these techniques have scalability problems for large systems.

2.4.2.5 Probabilistic approaches

Probabilistic approaches consist essentially of determining a probability of an event, for example, the probability of a class being a CS. Some techniques consist on the use of BBN, considering the CS detection process as a fuzzy-logic problem or frequent pattern tree.

2.4.2.6 Cooperative-based approaches

Cooperative-based CS techniques are primarily aimed at improving *Accuracy* and performance in CS detection. This is achieved by performing various activities cooperatively.

The only study that uses a cooperative approach is Boussaa et al. [S41]. According to the authors, the main idea is to evolve two populations in parallel. The first population through a metric-based approach creates a set of detection rules, maximizing the coverage of a set of CS, while the second population maximizes the number of artificially generated CS that are not covered by the detection rules of the first population [S41].

2.4.2.7 Manual approaches

Manual techniques are human-centric, tedious, time-consuming, and error prone. These techniques require a great human effort, therefore are not effective for detecting CS in large systems. According to the authors of [62], another important issue is that locating CS manually has been described as more a human intuition than an exact science.

The only study that uses a manual approach is [S52], where a catalog for the detection and handling of model smells for MATLAB / Simulink is presented. In this study, 3 types of techniques are used - manual, metric-based, symptom-based - according to the type of smell. The authors note that the detection of certain smells like the *Vague Name* or *Non-optimal Signal Grouping* can only be performed by manual inspection, because of the expressiveness of the natural language.

2.4.3 Dataset availability (F2)

The second finding is whether the underlying dataset is available - a precondition for study replication. When we talk about the dataset, i.e. the oracle, we are considering the software systems where CS and anti-patterns were detected, the type and number of CS and anti-patterns detected, and other data needed for the method used, e.g. if it is a metric-based approach the dataset must have the metrics for each application.

Only 12 studies provide a link to their dataset. However, 2 studies, [S28] and [S32], no longer have the active links. Thus, only 12.0% of the studies (10 out of 83, [S18, S27, S38, S51, S56, S59, S69, S70, S74, S82]) provide the dataset.

Another important feature for defining the dataset is which software systems are used in studies on which CS detection is performed. The number of software systems used in each study varies widely, and there are studies ranging from only one system to studies using 74 Java software systems and 184 Android apps with source code hosted in open source repositories. Most studies (83.1%) use open-source software. Proprietary software is used in 3.6% of studies and the use of the two types, open-source and proprietary, is used in 3.6% of studies. It should be noted that 9.7% of studies do not make any reference to the software systems being analysed.

Table 2.8: Top ten open-source software projects used in the studies

Open-source software	Nº of Studies	% Studies
Apache Xerces	28	33.7%
GanttProject	14	16.9%
ArgoUML	11	13.3%
Apache Ant	10	12.0%
JFreeChart	8	9.6%
Log4J	7	8.4%
Azureus	7	8.4%
Eclipse	7	8.4%
JUnit	5	6.0%
JHotDraw	5	6.0%

Table 2.8 presents the most used open-source software in the studies, as well as the number of studies where they are used and the overall percentage. *Apache Xerces* is the most used (33.7% of the studies), followed by *GanttProject* with 16.9%, *ArgoUML* with 13.3%, and *Apache Ant* used in 12.0% of the studies.

2.4.4 Programming language (F3)

In our research we do not make any restriction regarding the object-oriented programming language that supports the detection of CS. So, we have CS detection in 7 types of languages, in addition to the techniques that are language independent (3 studies) and a study [S66] that is for Android Apps without defining the type of language, as shown in Figure 2.5

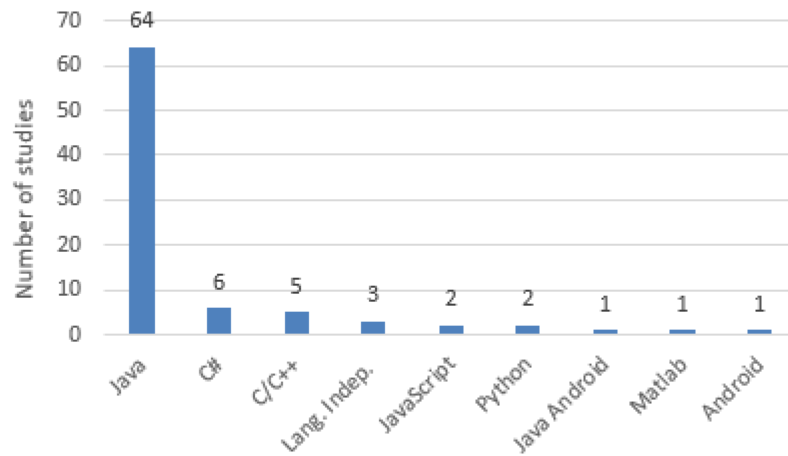


Figure 2.5: Programming languages and number of studies that use them

77.1% of the studies (64 out of 83) use Java as a target language for the detection of CS. C# is the second most used programming language, with 6 studies (7.2%), the third most used language is C/C++ with 5 studies (6.0%). JavaScript and Python are used in 2 studies (2.4%). Finally, we have 2 languages, MatLab and Java Android, which are used in only 1 study (1.2%). In total we found seven different types of program languages to be used as support for the detection of CS.

In our analysis we found that 3.6% of studies (3 out of 83, [S20, S32, S47]) present language-independent CS detection technique. When we related the studies that are language-independent with the used approach, we found that two of the three studies used Symptom-based and one the Metric-based approach. These results are in line with what was expected, since a symptom-based approach is the most susceptible of being adapted to different programming languages.

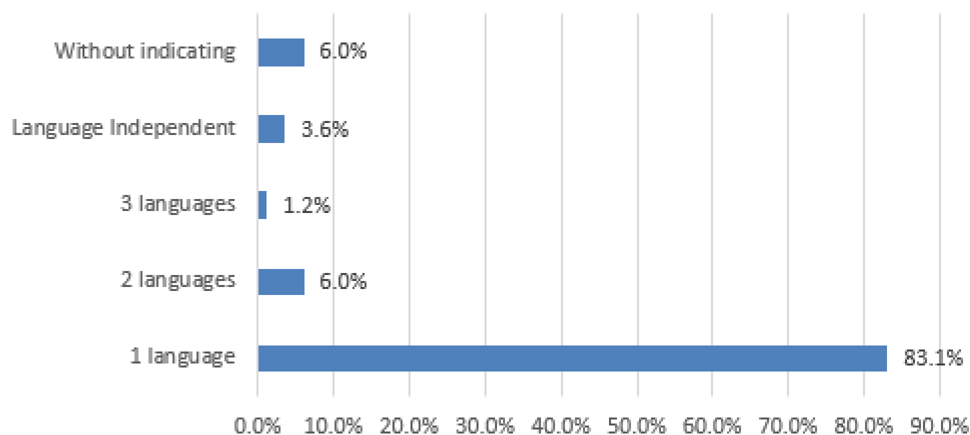


Figure 2.6: Number of languages used in each study

Multi-language support for CS detection is also very limited, as shown in Figure 2.6. In addition to the three independent language studies, only [S28], one study (1.2%) of the 83 analyzed, supports 3 programming languages. Five studies (6.0%, [S3, S9, S12, S15, S57]) detect CS in 2 languages and 69 studies (83.1%) only detect in a one programming language.

Five studies (6.0%) explain the detection technique, but do not refer to any language.

When we analyze the 5 studies that do not indicate any programming language, we find that all use a visualization-based approach, i.e., 41.7% (5 out of 12) of studies that use visualization techniques do not indicate any programming language.

2.4.5 Code smells detected (F4)

Several authors use different names for the same CS, so to simplify the analysis we have grouped the different CS with the same mean into one, for example, *Blob*, *Large Class* and *God Class* were all grouped in *God Class*. The description of CS can be found in Appendix A.4.

In Table 2.9 we can see the CS that are used in more than 3 studies, the number of studies in which they are detected and the respective percentage. As we have already mentioned in subsection 2.4.4, in this systematic review we do not make any restriction regarding the object-oriented programming language used. Thus, considering all object-oriented programming languages 68 different CS are detected, much more than the 22 described by Fowler [43]. *God Class* is the most detected CS, being used in 51.8 % of the studies, followed by *Feature Envy* and *Long Method* with 33.7 % and 26.5 %, respectively.

Table 2.9: Code smells detected in more than 3 studies

Code smell	**Nº of studies**	**% Studies**
God Class (Large Class or Blob)	43	51.8%
Feature Envy	28	33.7%
Long Method	22	26.5%
Data class	18	21.7%
Functional Decomposition	17	20.5%
Spaghetti Code	17	20.5%
Long Parameter List	12	14.5%
Swiss Army Knife	11	13.3%
Refused Bequest	10	12.0%
Shotgun Surgery	10	12.0%
Code clone/Duplicated code	9	10.8%
Lazy Class	8	9.6%
Divergent Change	7	8.4%
Dead Code	4	4.8%
Switch Statement	4	4.8%

All 68 CS detected are listed in Appendix A.5, as well as the number of studies in which they are detected, their percentage, and the programming languages in which they are detected.

When we analyzed the CS detected in each study, we found that the number is low, with an average of 3.6 CS per study and a mode of 1 CS. Only the study [S57], with a symptom-based approach, detects the 22 CS described by Fowler [43].

Figure 2.7 shows the number of CS detected by number of studies. We can see that the number of smells most detected is 1 (in 24 studies), 4 smells are detected in 13 studies and 11 studies detect 3 smells. The detection of 12, 13, 15 and 22 smells is performed in only 1 study. It should be noted that 5 studies do not indicate which CS they detected. It is also important to note that these 5 studies use a visualization-based approach.

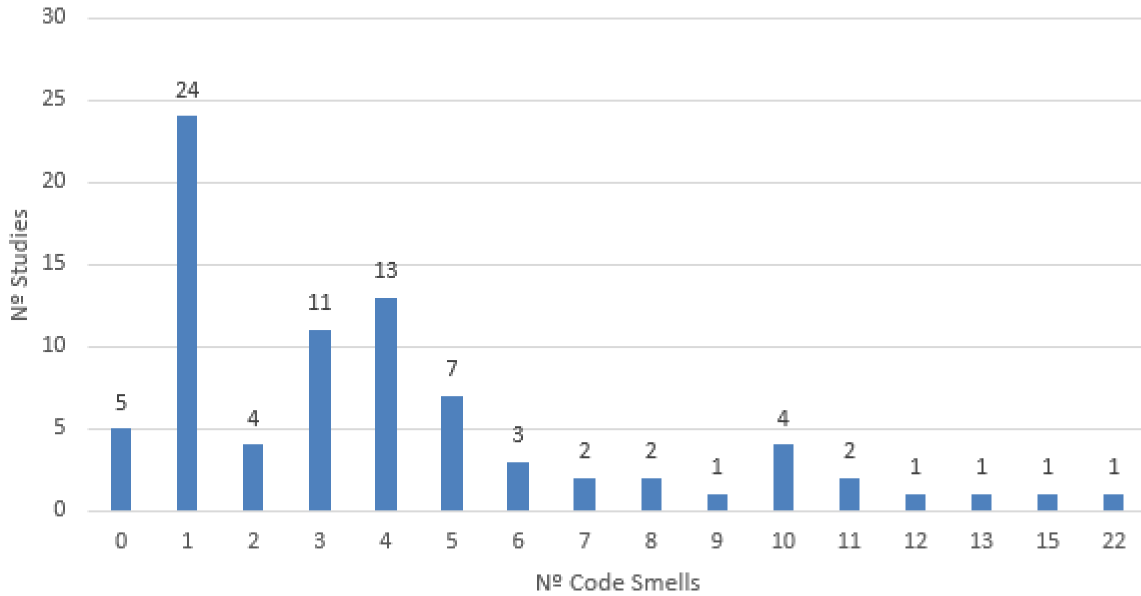


Figure 2.7: Number of code smells detected by number of studies

2.4.6 Machine Learning techniques used (F5)

ML algorithms present many variants and different parameter configurations, making it difficult to compare them. For example, the Support Vector Machines algorithm has variants such as SMO, LibSVM, etc. Decision Trees can use various algorithms such as C5.0, C4.5 (J48), CART, ID3, etc. As algorithms are presented with different details in the studies, for a better understanding of the algorithm used, we classify ML algorithms in their main category, creating 9 groups as shown in the table 2.10.

Table 2.10 shows the ML algorithms, the number of studies using the algorithm and its percentage, as well as the ID of the studies that use the algorithm.

From the 83 primary studies analyzed 35% of the studies (29 out of 83, [S6, S18, S22, S24, S27, S28, S31, S33, S34, S36, S40, S41, S42, S43, S45, S50, S53, S54, S56, S58, S64, S66, S70, S71, S74, S75, S77, S79, S83]) use ML techniques in CS detection. Except for 3 studies [S36, S56, S77] where multiple ML algorithms are used, the other 26 studies use only 1 algorithm in CS detection.

The most widely used algorithms are Genetic algorithms (9 out 83, [S31, S41, S43, S45, S58, S64, S66, S70, S79]) and Decision Trees (8 out 83, [S6, S28, S36, S40, S53, S56, S77, S83]), which are used in 10.8% and 9.6%, respectively, of the analyzed studies. We think that a possible reason why genetic algorithms are the most used algorithm, is because they are used to generate the CS detection rules and to find the best threshold values to be used in the detection rules.

Table 2.10: ML algorithms used in the studies

ML algorithm	Nº of Studies	% Studies	Studies
Genetic Programming	9	10.8%	S31, S41, S43, S45, S58, S64, S66, S70, S79
Decision Tree	8	9.6%	S6, S28, S36, S40, S53, S56, S77, S83
Support Vector Machines (SVM)	6	7.2%	S33, S34, S36, S56, S71, S77
Association Rules	6	7.2%	S36, S42, S50, S54, S56, S77
Bayesian Algorithms	5	6.0%	S18, S27, S36, S56, S77
Random Forest	3	3.6%	S36, S56, S77
Neural Network	2	2.4%	S74, S75
Regression models	1	1.2%	S22
Artificial Immune Systems (AIS)	1	1.2%	S24

Regarding Decision trees, it is due to the easy interpretation of the models, mainly in its variant C4.5 / J48 / C5.0.

The third most used algorithms for ML, used in 7.2% of the studies, are Support Vector Machines (SVM) (6 out 83, [S33, S34, S36, S56, S71, S77]) and association rules (6 out 83, [S36, S42, S50, S54, S56, S77]) with Apriori and JRip being the most common.

Bayesian Algorithms are the fifth most used algorithm with 6.0% (5 out 83, [S18, S27, S36, S56, S77]).

The other 4 ML algorithms that were also used are Random Forest (in 3 studies), Neural Network (in 2 studies), Regression models (in 1 study) and Artificial Immune Systems (AIS) (in 1 study).

2.4.7 Evaluation of techniques (F6)

The evaluation of the techniques used is an important factor to realize their effectiveness and consequently choose the best technique or tool. The main metrics used to evaluate the techniques are *Accuracy*, *Precision*, *Recall*, and *F-Measure*. These 4 metrics are calculated based on true positives (TP), false positives (FP), false negatives (FN), and true negatives (TN) instances of CS detected, according to the following formulas:

- *Accuracy* = (TP + TN) / (TP + FP + FN + TN)
- *Precision* = TP / (TP + FP)
- *Recall* = TP / (TP + FN)
- *F-Measure* = 2 * (*Recall* * *Precision*) / (*Recall* + *Precision*)

In the 83 articles analyzed, 86.7% (72 studies) evaluated the technique used and 13.3% (11 studies) did not. Table 2.11 shows the most used evaluation metrics. *Precision* is the most used with 46 studies (55.4%), followed by *Recall* in 44 studies (53.0%) and *F-Measure* in 17 studies (20.5%). It should be noted that 28 studies (33.7 %) use other metrics for evaluation such as the number of detected defects, *Area Under Receiver Operating Characteristic (ROC)*, *Standard Error*

(SE) and *Mean Square Error* (MSE), *Root Mean Squared Prediction Error* (RMSPE), *Prediction Error* (PE), etc.

Table 2.11: Metrics used to evaluate the detection techniques

Metric	Nº of studies	% Studies
Precision	46	55.4%
Recall	44	53.0%
F-measure	17	20.5%
Accuracy	10	12.0%
Other	28	33.7%
Without evaluation	11	13.3%

In the last years the most used metrics in the evaluation are *Precision* and *Recall*, but until 2010 few studies have evaluations based on these metrics, presenting only the CS detected. When we analyze the evaluations of the different techniques, we verified that the results depend on the applications used to create the oracle and the CS detected, so we have several studies that have chosen to present the means of *Precision* and *Recall*.

Regarding the different approaches used, we can conclude that:

1. in manual approaches and cooperative-based approaches, since we only have one study for each, we cannot draw conclusions;
2. in the visualization-based approaches, most of the evaluations presented are qualitative, and almost half of the studies do not present an evaluation;
3. in relation to the other 4 approaches (probabilistic, metric, symptom-based, and search-based), all present at least one study/technique with 100% *Recall* and *Precision* results.
4. It is difficult to make comparisons across the different techniques since, except for the studies of the same author(s), all the others have different oracles.

The usual way to build an oracle is to choose a set of software systems (typically open source), choose the CS that you want to detect and ask a group of MSc students (3, 4 or 5 students), supervised by experts (e.g. software engineers), to identify the occurrences of smells in systems. In case of doubt on a candidate CS, either the expert decides, or the group reaches a consensus on whether this candidate is, or not, a CS. As you can see, the creation of an oracle is not an easy task, because it requires a considerable amount of manual work in the detection of CS, encompassing the aforementioned problems of a manual approach (see section 2.4.2.7) mainly its subjectivity.

For a rigorous comparison of the evaluation of the different techniques, it is necessary to use common oracle (see section 2.4.3), which does not happen today.

2.4.8 Detection tools (F7)

Comparing the results of CS detection tools is important to understand the performance of the techniques associated with the tool and consequently to know which one is the best. It is also important to create tools that allow us to replicate studies.

When we analyzed which studies created a detection tool, we found that 61.4% (51 out of 83) studies developed a tool, as show in table 2.12.

Table 2.12: Number of studies that developed a tool and its approach

Approaches	**Nº studies**	**Nº studies with tool**	**% Studies in the approach**	**% Studies in total**
Symptom-Based	16	13	81.3%	15.7%
Metric-Based	20	13	65.0%	15.7%
Visualization-Based	12	10	83.3%	12.0%
Search-Based	25	9	36.0%	10.8%
Probabilistic	10	6	60.0%	7.2%
Cooperative-Based	1	0	0.0%	0.0%
Manual	1	0	0.0%	0.0%

The *Symptom-Based* and *Metrics-Based* approaches are those that present most of the developed tools with 15.7% (13 out of 83 studies), follow by *Visualization-Based* with 12.0% (10 out of 83 studies) (see Table 2.12). On the opposite side, there is the *Probabilistic* approach where only 7.2% (6 out of 83 studies) present developed tools.

When we analyze the percentage of studies that develop tools within each approach, we find that *Visualization-Based* and *Symptom-Based* approaches are those that have a greater number of developed tools with 83.3% (10 out of 12 studies) and 81.3% (13 out of 16 studies), respectively (see Table 2.12).

On the other side, there is the *Search-Based* approach where only 36.0% (9 out of 25 studies) present developed tools. In this approach, less than half of the studies present a tool because authors used existing tools instead of creating new ones. For example, some studies [S34, S36, S56, S77] use Weka [4] to implement their techniques.

As Rasool and Arshad mentioned in their study [109], it becomes arduous to find common tools that performed experiments on common systems for extracting common smells. Different techniques perform experiments on different systems and present their results in different formats. When analyzing the results of different tools to verify their results, examining the same software packages and CS, we verified a disparity of results [37, 109].

2.4.9 Thresholds definition (F8)

Threshold values are a very important component in some detection techniques because they are the values that define whether or not a candidate is a CS. Its definition is very complicated and one of the reasons why there is so much disparity in the detection results of CS (see section 2.4.8). Some studies use genetic algorithms to calibrate threshold values as a way of reducing subjectivity in CS detection, e.g. [S70].

A total of 44 papers use thresholds in their detection technique, representing 53.0% of all studies. 47.0% of studies (39 out of 83 studies) did not use thresholds.

[4] Weka is a collection of ML algorithms for data mining tasks (www.cs.waikato.ac.nz/ml/weka/)

Table 2.13: Number of studies that use thresholds in CS detection

Approaches	Nº studies	Nº use thresholds	% Studies in the approach	% Studies in total
Metric-Based	20	15	75.0%	18.1%
Symptom-Based	16	11	68.8%	13.3%
Search-Based	25	8	32.0%	9.6%
Probabilistic	10	8	80.0%	9.6%
Visualization-Based	12	1	8.3%	1.2%
Cooperative-Based	1	1	100.0%	1.2%
Manual	1	0	0.0%	0.0%

Without the *Cooperative-Based* approach (which presents only 1 study), in the total of studies, *Metric-Based* and *Symptom-Based* approaches are those that present the most used of thresholds with 18.1% (15 out of 83 studies) and 13.3% (11 out of 83 studies), respectively (see Table 2.13).

When we analyze the number of studies within each approach that uses thresholds, we find that the three approaches that most use thresholds in their detection techniques are Probabilistic with 80% (8 out of 10 studies), *Metric-Based* with 75.0% (15 out of 20 studies), and *Symptom-Based* with 68.8% (11 out of 16 studies). In *visualization-Based* approaches, only one study use thresholds in their CS detection techniques, as shown in Table 2.13.

Analyzing the detection techniques, we verified that these results are in line with what was expected, since we found that the *Probabilistic* and *Metric-Based* approaches are those that most need to use thresholds. In the *Probabilistic* approaches thresholds are required to define the values of support, confidence and probabilistic decision values. In *Metric-Based* approaches, it is essential to define threshold values for the different metrics that compose the rules.

2.4.10 Validation of techniques (F9)

The validation of a technique is performed by comparing the results obtained by the technique, with the results obtained through another technique with similar objectives. Obviously, both techniques must detect the same CS in the same software systems. The most usual forms of validation are: using the techniques of various existing approaches, such as manuals; use existing tools; comparing the results with those of other published papers.

When we analyze how many studies are validating their technique (see Table 2.14), we verified that 62.7% (52 out of 83) of the studies do not validate the technique.

Considering the differences between techniques and all subjectivity in a technique (see sections 2.4.9, 2.4.8, 2.4.7), we conclude that it is not easy to perform validations with tools that implement other techniques, even if they have the same goals. Thus, it is not surprising that one of the most common methods for validating the results is express opinion, with a percentage of 26.9% of the studies (14 of the 52 studies doing validation, [S3, S6, S8, S13, S15, S23, S31, S38, S43, S44, S53, S54, S56, S66]), as shown in Table 2.14.

Some authors, as in [S8], claim that validation was performed by experts because only

Table 2.14: Tools / approach used by the studies for validation

Tool/approach	Nº of Studies	% Studies	Studies
DECOR	14	26.9%	S18, S24, S27, S30, S31, S42, S43, S45, S48, S50, S51, S59, S62, S79
Manually	14	26.9%	S3, S6, S8, S13, S15, S23, S31, S38, S43, S44, S53, S54, S56, S66
JDeodorant	10	19.2%	S42, S44, S45, S50, S51, S54, S62, S72, S73, S75
iPlasma	8	15.4%	S26, S49, S53, S54, S56, S65, S68, S81
Machine Learning	7	13.5%	S24, S43, S58, S66, S67, S79, S83
Papers	6	11.5%	S41, S51, S60, S61, S74, S77
DETEX	3	5.8%	S33, S34, S71
Incode	3	5.8%	S53, S64, S44
inFusion	3	5.8%	S65, S53, S49
PMD	3	5.8%	S49, S56, S53
BDTEX	2	3.8%	S33, S59
CodePro AnalytiX	2	3.8%	S55, S78
Jtombstone	2	3.8%	S55, S78
Rule Marinescu	2	3.8%	S47, S65
AntiPattern Scanner	1	1.9%	S56
Bellon benchmark	1	1.9%	S15
Checkstyle	1	1.9%	S49
DCPP	1	1.9%	S50
DUM-Tool	1	1.9%	S78
Essere	1	1.9%	S68
Fluid Tool	1	1.9%	S56
HIST	1	1.9%	S42
JADET	1	1.9%	S20
Jmove	1	1.9%	S75
JSNOSE	1	1.9%	S70
Ndepend	1	1.9%	S73
NiCad	1	1.9%	S28
SonarQube	1	1.9%	S50

maintainers can assess the presence of defects in design depending on their design choices and in the context, or as in [S23] where validation was performed by independent engineers who assessed whether suspicious classes are smells, depending on the context of the systems studied/analysed.

Equally with manual validation is the use of the DECOR tool [87], also used in 26.9% of studies (14 out of 52, [S18, S24, S27, S30, S31, S42, S43, S45, S48, S50, S51, S59, S62, S79]), this approach is based on symptoms. DECOR is a tool proposed by Moha et al. [87] which uses a Domain-Specific Language (DSL) to describe CS. They used this DSL to describe well-known smells, *Blob* (aka *Long Class*), *Functional Decomposition*, *Spaghetti Code*, and *Swiss Army Knife*. They also presented algorithms to parse rules and automatically generate detection algorithms.

The following two tools most used in validation, with 19.2% (10 out of 52 studies) are JDeodorant [39] used for validation of the studies [S42, S44, S45, S50, S51, S54, S62, S72, S73,

S75], and iPlasma [81] for the studies [S26, S49, S53, S54, S56, S65, S68, S81]. JDeodorant [5] is a plug-in for the *Eclipse IDE* developed by Fokaefs et al. for automatic detection of some CS (*God Class*, *Type Check*, *Feature Envy*, *Long Method*) and performs refactoring. iPlasma [6] is a tool that uses a metric-based approach to CS detection developed by Marinescu et al.

Seven studies [S24, S43, S58, S66, S67, S79, S83] compare their results with the results obtained through ML techniques, namely Genetic Programming (GP), BBN, and Support Vector Machines (SVM). The ML techniques represent 13.5% of the studies (7 out of 52) that perform validation.

As we can see in table 2.14, where we present 28 different ways of doing validation, there are still many other tools used to validate detection techniques.

2.4.11 Replication of the studies (F10)

The replication of a study is an important process in Software Engineering, and its importance is highlighted by several authors such as Shull et al. [121] and Barbara Kitchenham [67]. According to Shull et al. [121], replication helps to *"better understand software engineering phenomena and how to improve the practice of software development. One important benefit of replications is that they help mature software engineering knowledge by addressing both internal and external validity problems.*"The same authors also mention that in terms of external validation, replications help to generalize the results, demonstrating that they do not depend on the specific conditions of the original study. In terms of internal validity, replications also help researchers show the range of conditions under which experimental results hold. These authors still identify two types of replication: exact replications and conceptual replications.

Another author to emphasize the importance of replication is Kitchenham [67], claiming that "replication is a basic component of the scientific method, so it hardly needs to be justified."

Given the importance of replication, it is important that studies provide the necessary information to enable replication. Especially in exact replications, where the procedures of an experiment are followed as closely as possible to determine if the same results can be obtained [121]. Thus, our goal is not to perform replications, but to verify that the study has the conditions to be replicated.

According to Carver [24] [25], a replication paper should provide the following information about the original study (at a minimum): Research questions, Participants, Design, Artifacts, Context variables, Summary of results. This information about the original study is required to provide sufficient context to understand replication. Thus, we consider that for a study to be replicated, it must provide the aforementioned information. For CS detection studies the relevant artefacts are the target Software Systems (usually open-source, as seen in section 2.4.3), the oracles and the CS collection tool/instrument.

Oracles are extremely important for the replication of CS detection studies. However, since oracles are usually built based on expert's opinion, they are one of the major sources of subjectivity in CS detection.

[5] https://users.encs.concordia.ca/ nikolaos/jdeodorant/
[6] http://loose.utt.ro/iplasma/

As we have seen in section 2.4.3, only 10 studies present the available dataset, providing a link to it, however 2 studies, [S28] and [S32], no longer have the active links. Thus, only 12.0% of the studies (10 out of 83, [S18, S27, S38, S51, S56, S59, S69, S70, S74, S82]) makes the dataset available, and are this candidates for replication.

Another of the important information for the replication is the existence of an artifact, it happens that the studies [S70] amd [S74] does not present an artifact, therefore cannot be replicated.

We conclude that only 9.6% of the studies (8 out of 83, [S18, S27, S38, S51, S56, S59, S69, S82]) can be replicated according to Carver requirements [24].

It is noteworthy that [S51] makes available on the Internet a replication package composed of Oracles, Change History of the Object systems, Identified Smells, Object systems, Additional Analysis - Evaluating the HIST with *Cassandra* Releases.

2.4.12 Visualization techniques (F11)

CS visualization can be approached in two different ways, (i) CS detection is done through a non-visual approach and the visualization shows the CS in the code, or (ii) the CS detection is performed through a visual approach.

Regarding the first approach, we found 5 studies [S9, S40, S57, S72, S81], corresponding to 6.0% of the studies analyzed in this SLR. Thus, we can conclude that most studies are only dedicated to detecting CS, but do not pay much attention to visualization. Most of the proposed CS visualization shows the CS inside the code itself. This approach works for some systems, but when we are in the presence of large legacy systems, it is too detailed for a global refactoring strategy. Thus, a more macro approach is required, without losing detail, to present CS in a more aggregated form.

Regarding to the second approach, where a visualization-based approach is used to detect CS, it represents 14.5% of the studies (12 out of 83, [S1, S2, S7, S11, S12, S16, S17, S19, S21, S39, S46, S76]). One of the problems pointed to the visualization-based approach is the scalability for large systems, since this type of approach is semi-automatic, requiring human intervention. Nevertheless, we found 3 studies [S7, S17, S16] with solutions dedicated to large systems.

Most studies providing visualization feature show the system structure in packages, classes, and methods. We could not find examples where view were adapted to the type of CS,for example, in a CS detection context it is not necessary to show the parts of the software where there are no CS, since it is only adding unnecessary complexity.

Combining the two types of approaches, we conclude that 20.5% of the studies (17 out of 83) use some kind of visualization in their approach.

As for code smells coverage, the *Duplicated Code* CS (aka *Code Clones*) is definitely the one where more visualization techniques have been applied. *Recall* from Section 2.2 that Zhang et al. [146] systematic review on CS revealed that *Duplicated Code* is the most widely studied CS. Also in that section, we referred to Fernandes et al. [37] systematic review that concluded that *Duplicated Code* was among the top-three CS detected by tools, due to its importance. The application of visualization to the *Duplicated Code* CS ranges from 2D techniques (e.g. dot plots / scatterplots, wheel views / chord diagrams, and other graph-based and polymetric view-based

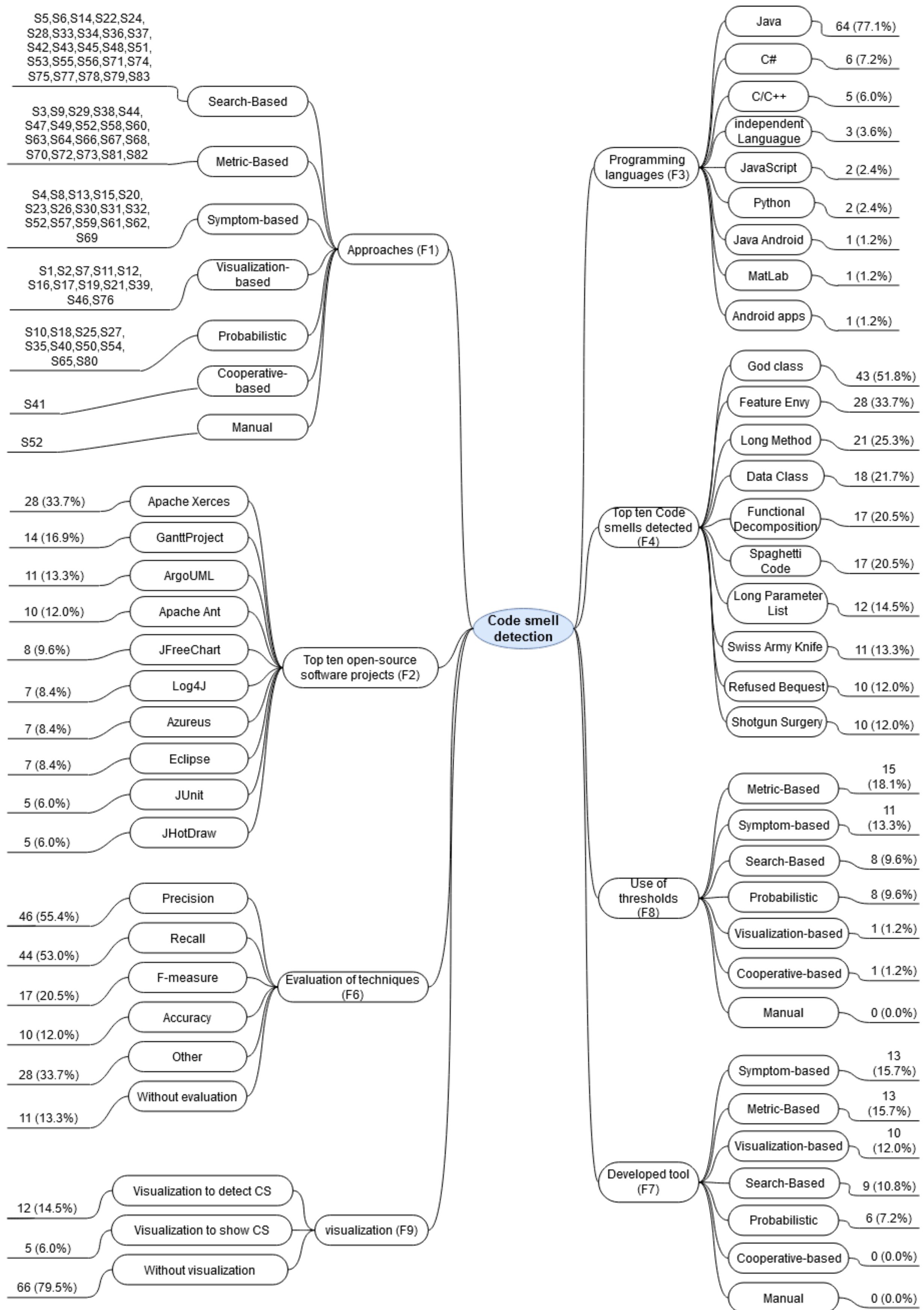


Figure 2.8: Summary of main findings

ones) to more sophisticated techniques, such as those based on 3D metaphors and virtual reality. A comprehensive mapping study on the topic of *Duplicated Code* visualization has just been published [52].

2.5 Discussion

We now address our research questions, starting by discussing what we found for each of them, mainly addressing the benefits and limitations of evidence of these findings. The mind map in Figure 2.8 provides a summary of main findings. Finally, we discuss the validation and limitations of this systematic review.

2.5.1 Research Questions (RQ)

This subsection aims to discuss the answers to the three research questions and how the findings and selected documents addressed these issues. In figure 2.2 we show the selected studies and the respective research questions they focus on. Regarding how findings (**F**) interrelate with research questions, findings F1, F2, F3, F4, F5 support the answer of RQ1, findings F5, F6, F7, F8, F9, F10, support the answer of RQ2, and, finally, F11 supports the answer of RQ3 (see figure 2.9).

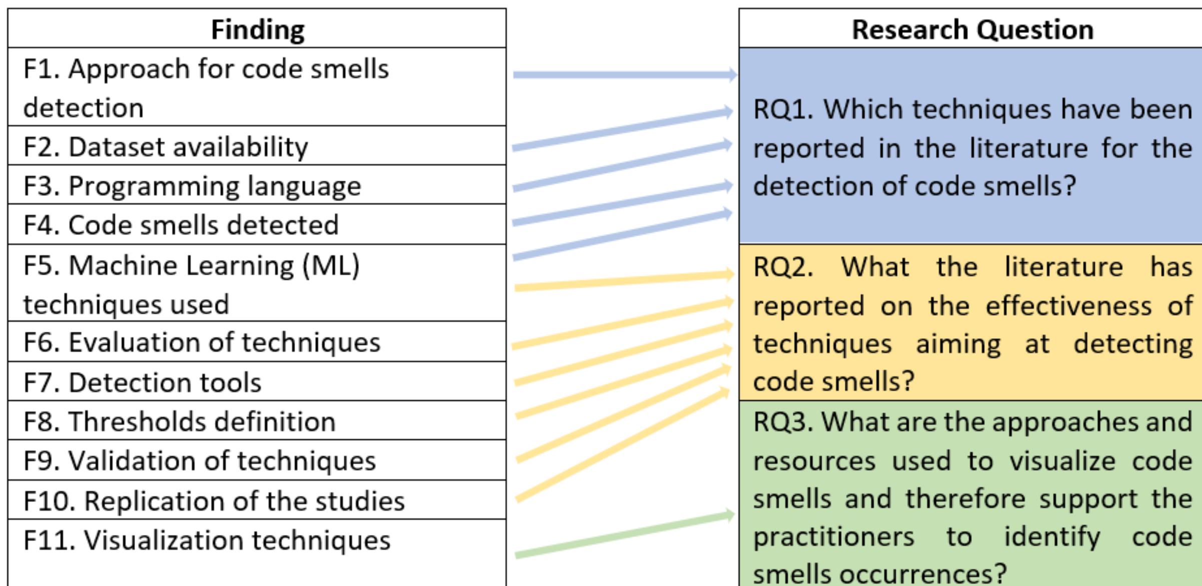


Figure 2.9: Relations between findings and research questions

RQ1. *Which techniques have been reported in the literature for the detection of CS?*

To answer this research question, we classified the detection techniques in seven categories, following the classification presented in Kessentini et al. [62] (see F1, subsection 2.4.2). The Search-Based approach is applied in 30.1% studies. These types of approaches are inspired by contributions in the domain of Search-Based Software Engineering (SBSE) and most techniques in this category apply ML algorithms, with special incidence in the algorithms of genetic programming, decision trees, and association rules. The second most used approach is the

Metric-Based applied in 24.1% studies. The Metric-Based approach consists in the use of rules based on a set of metrics and respective thresholds to detect specific CS. The third most used approach, with 19.3% of studies, is the Symptom-Based approach. It consists of techniques that describe the symptoms of CS and later translate these symptoms into detection algorithms.

Regarding datasets (see F2, subsection 2.4.3), more specifically the oracles used by the different techniques, we concluded that only 10 studies (12.0% of studies) provided them. Most studies, 83.1%, use open-source software, the most used systems being Apache Xerces (33.7% of studies), GanttProject (16.9%), and ArgoUML with 13.3%.

Regarding the programming language used in the target software systems, Java is used in 64 studies (77.1%) of the studies use Java as a support language for the detection of CS. C# and C++ are the other two most commonly used programming languages in CS detection, being used in 7.2% and 6.0% of studies respectively (see F3, subsection 2.4.4). Multi-language support for CS detection is also very limited, with the majority (83.1%) of the studies analyzed supporting only one language. The maximum number of supported languages is three, being reported exclusively in [S28].

Regarding the most detected CS, *God Class* stands out with 51.8% (see F4, subsection 2.4.5). *Feature Envy* and *Long Method* with 33.7% and 26.5%, respectively, are the next two most commonly used CS. When we analyze the number of CS detected by each study, we found that they detected on average 3 CS, but the most frequent case is the studies detect only 1 CS.

RQ2. *What literature has reported on the effectiveness of techniques aiming at detecting CS?*

Finding out the most effective technique to detect CS is not a trivial task. We have realized that all identified approaches have pros and cons, presenting factors that can bias the results. Although in the evaluation of the techniques (F6, subsection 2.4.7) we found that for 4 approaches (Probabilistic, Metric-Based, Symptom-Based, and Search-Based) there are reported techniques with 100% *Accuracy* and *Recall* results, the detection problem is very far to be solved. These results only apply to the detection of simpler CS, e.g. *God Class*, so it is not surprising that 51.8% of the studies use this CS, as we can see in table 2.9. In relation to the more complex CS, the results are much lower and very few studies use them. The fact that only one study [S57] detects the 22 CS described by Fowler [43] is noteworthy.

The answer to RQ2 is that there is not just one technique, but several techniques, depending on several factors such as:

- Code smell to detect - We found that there is no technique that generalizes to all CS. When we analyzed the studies that detected the greatest number of CS, [S38, S53, S57, S63, S69] (more than 10 CS), we find that *Precision* and *Recall* depend largely on the CS.

- Software systems - The same technique when detecting the same CS in different software systems, presents a great discrepancy in the number of false positives and false negatives and consequently in *Precision* and *Recall*.

- Threshold values - There is no consensus regarding the definition of threshold values. The variation of this value causes more, or less, CS to be detected, thus varying the number of false positives. Some authors try to define thresholds automatically, namely using genetic programming algorithms.

- Oracle - Oracles are a key part of most CS detection processes, for example, in the training of ML algorithms, and are fundamental for replication. However, there is no widespread practice of oracle sharing.

Regarding the automation of CS detection processes, thus making them independent of thresholds, we found that 35% of the studies used ML techniques. However, when we look at how many of these studies do not require thresholds, we find that only 18.1% (15 out of 83, [S22, S33, S34, S36, S40, S43, S53, S56, S66, S71, S74, S75, S77, S79, S83]) are truly automatic.

RQ3. *What are the approaches and resources used to visualize CS and therefore support the practitioners to identify CS occurrences?*

The visualization and representation are of extreme importance, considering the variety of CS, possibilities of location in code (within methods, classes, between classes, etc.), and dimension of the code for a correct identification of the CS. Unfortunately, most of the studies do not visually represent the detected CS. Within the 83 selected studies in the SLR, only 5 out of the 71 that do not use Visualization-Based detection, visually represent the detected CS.

In the 14.5% studies (12 out of 83) that use visualization-based approaches to detect CS, several methods are used to show the structure of the programs, such as: (1) city metaphors [S7, S17]; (2) 3D visualization technique [S16, S17]; (3) interactive ambient visualization [S19, S39, S46]; (4) multiple views adapted to CS [S12,S21]; (5) polymetric views [S2, S46]; (6) graph model [S1]; (7) multivariate visualization techniques, such as parallel coordinates, and non-linear projection of multivariate data onto a 2D space [S76]; (8) in [S46] several views are displayed such as node-link-based dependency graphs, grids and spiral egocentric graphs, and relationship matrices.

With respect to large systems, only three studies present dedicated solutions [S7, S17, S16].

2.5.2 SLR validation

To ensure the reliability of the SLR, we carried out validations in 3 stages:

i) The first validation was carried out in the application of the inclusion and exclusion criteria, through the application of Fleiss' Kappa [38]. Through this statistical measure, we validated the level of agreement between the researchers in the application of the inclusion and exclusion criteria.

ii) The second validation was carried out through a focus group, when the quality criteria were applied in stage 4.

iii) To validate the results of the SLR we conducted 3 surveys, each divided into 2 parts, one on CS detection and another on CS visualization. Each question in the surveys consisted of 3 parts: 1) the question itself about one of the findings being evaluated on a 6 point Likert scale (Strong disagreement, Disagreement, Weak disagreement, Agreement, Strong agreement); 2) a slider between 0 and 4 measuring the degree of confidence of the answer; 3) an optional field to describe the justification of the answer or for comments.The three inquiries were intended to:

1) Pre-test, with the aim of identifying unclear questions and collecting suggestions for improvement. The subjects chosen for the pre-test were Portuguese researchers with the most relevant work in the area of software engineering, totaling 27;

2) The subjects in the second survey were the authors of the studies that are part of this SLR, totaling 193;

3) The third survey was directed at the software visualization community; we chose the authors from all papers selected for the SLR on software visualization by Merino et al.[85] that were taken exclusively from the SOFTVIS and VISSOFT conferences, totaling 380; we also distributed this survey through a post on a Software Visualization blog[7]. The surveys were deployed in the Qualtrics Online Plataform [8].

The structure of the surveys, collected responses, and descriptive statistics on the latter are available at a github repository[9] and Zenodo [111].

In table 2.15 we present a summary of the results of the responses from this SLR' authors (2nd survey) and from the visualization community (3rd survey). As we can see, using the aforementioned scale, most participants agree with SLR results. The grayed cells in this table represent, for each finding, the answer(s) that obtained the highest score. We can then observe that: 10% of the findings had *Strong agreement* as its higher score, 80% of the findings had *Agreement* and, 20% had *Weak agreement*.

Regarding the question, *Please select the 3 most often detected code smells?*, the answers placed the *Long Method* as the most detected CS, followed by *God Class* and *Feature Envy*. In our SLR, based on actual data, we concluded that the most detected CS is *God Class*, followed by *Feature Envy* and *Long Method*. This mismatch is small, since it only concerns the relative order of those 3 code smells, and shows that the community is well aware of which are the most often detected CS.

2.5.3 Validity threats

We now go through the types of validity threats and corresponding mitigating actions that were considered in this study.

Conclusion validity. We defined a data extraction form to ensure consistent extraction of relevant data for answering the research questions, therefore avoiding bias. The findings and implications are based on the extracted data.

Internal validity. To avoid bias during the selection of studies to be included in this review, we used a thorough selection process, comprised of multiple stages. To reduce the possibility of missing relevant studies, in addition to the automatic search, we also used snowballing for complementary search.

External validity. We have selected studies on code smells detection and visualization. The exclusion of studies on related subjects (e.g. refactoring and technical debt) may have caused some studies also dealing with code smells detection and visualization not to be included. However, we have found this situation to occur in breadth papers (covering a wide range of topics) rather than in depth ones (covering a specific topic). Since the latter are the more important ones for primary studies selection, we are confident on the relevance of the selected sample.

[7] https://softvis.wordpress.com/

[8] https://www.qualtrics.com/

[9] https://github.com/dataset-cs-surveys/Dataset-CS-surveys.git

Table 2.15: Summary of survey results

Answer / Question(finding)	Strong agreement	Agreement	Weak agreement	Weak disagreement	Disagreement	# of answers	Resp. confidence degree (1-4) Average	Std. deviation
The most frequently used CS detection techniques are based on rule-based approaches (F1)	35.3%	47.1%	11.8%	5.9%	0.0%	34	3.2	0.8
Very few CS detection studies provide their oracles (a tagged dataset for training detection algorithms) (F2)	26.5%	58.8%	11.8%	2.9%	0.0%	34	3.1	0.7
In the detection of simpler CS (e.g. *Long Method* or *God Class*), the achieved *Precision* and *Recall* of detection techniques can be very high (up to 100%) (F6)	11.8%	44.1%	26.5%	0.0%	14.7%	34	3.2	0.5
When the complexity of CS is greater (e.g. *Divergent Change* or *Shotgun Surgery*), the *Precision* and *Recall* in detection are much lower than in simpler CS (F6)	11.8%	47.1%	26.5%	8.8%	5.9%	34	3.1	0.7
There are few oracles (a tagged dataset for training detection algorithms) shared and publicly available. The existence of shared and collaborative oracles could improve the state of the art in CS detection research (F2)	60.0%	34.3%	2.9%	2.9%	0.0%	35	3.6	0.5
The vast majority of CS detection studies do not propose visualization features for their detection (F11)	15.4%	66.7%	10.3%	5.1%	2.6%	39	3.0	1.0
The vast majority of existing CS visualization studies did not present evidence of its usage upon large software systems (F11)	12.5%	43.8%	34.4%	6.3%	0.0%	32	2.9	0.9
Software visualization researchers have not adopted specific visualization related taxonomies (F11)	9.4%	28.1%	46.9%	9.4%	6.3%	32	2.0	1.2
If visualization related taxonomies were used in the implementation of CS detection tools, that could enhance their effectiveness (F11)	11.8%	38.2%	38.2%	5.9%	5.9%	34	2.8	1.1
The combined use of collaboration (among software developers) and visual resources may increase the effectiveness of CS detection (F11)	23.5%	50.0%	26.5%	0.0%	0.0%	34	3.2	0.8

Construct validity. The studies identified from the systematic review were accumulated from multiple literature databases covering relevant journals and proceedings. In the selection process the first author made the first selection and the remaining ones verified and confirmed it. To avoid bias in the selection of publications we specified and used a research protocol including the research questions and objectives of the study, inclusion and exclusion criteria, quality criteria, search strings, and strategy for search and data extraction.

2.6 Conclusion

2.6.1 Conclusions on this SLR

This Systematic Literature Review has a twofold goal: the first is to identify the main CS detection techniques, and their effectiveness, as discussed in the literature, and the second is to analyze to which extent visual techniques have been applied to support practitioners in daily activities related to CS. For this purpose, we have specified 3 research questions (RQ1 through RQ3).

We applied our search string in six repositories (ACM Digital Library, IEEE Xplore, ISI Web of Science, Science Direct, Scopus, Springer Link) and complemented it with a manual search (backward snowballing), having obtained 1883 papers in total. After removing the duplicates, applying the inclusion and exclusion criteria, and quality criteria, we obtained 83 studies to analyze. Most of the studies were published in conference proceedings (76%), followed by journals (23%), and books (1%). The 83 studies were analysed on the basis of 11 points (findings) related to the approach used for CS detection, dataset availability, programming languages supported, CS detected, evaluation of techniques, tools created, thresholds, validation and replication, and use of visualization techniques.

Regarding RQ1, we conclude that the most frequently used detection techniques are based on search-based approaches, which mainly apply ML algorithms, followed by metric-based approaches. Very few studies provide the oracles used and most of them target open-source Java projects. The most commonly detected CS are *God Class*, *Feature Envy* and *Long Method*, by this order. On average, each study detects 3 CS, but the most frequent case is detecting only 1 CS.

As for RQ2, in the detection of simpler CS (e.g. *God Class*) 4 approaches are used (probabilistic, metric-based, symptom-based, and search-based) and authors claim to achieve 100% *Precision* and *Recall* results. However, when the complexity of CS is greater, the results have much lower relevance and very few studies use them. Thus, the detection problem is very far to be solved, depending on the detection results of the CS used, of the software systems in which they are detected, of the threshold and oracle values.

Regarding RQ3, we found that most studies that detect CS do not put forward a corresponding visualization feature. Several visualization approaches have been proposed for representing the structure of programs, either in 2D (e.g. graph-based, polymetric views) or in 3D (e.g. city metaphors), where the objective of allowing to identify potentially harmful design issues is claimed. However, we only found three studies that proposed dedicated solutions for CS visualization in large systems.

2.6.2 Open issues

Detecting and visualizing CS are nontrivial endeavors. While producing this SLR we obtained a comprehensive perspective on the past and ongoing research in those fields, that allowed the identification of several open research issues. We briefly overview each of those issues to set up a baseline for our own research presented in this thesis and also in the expectation it may inspire new researchers in the field.

(1) Code smells subjective definitions hamper a shared interpretation across researchers' and practitioners' communities, thus delaying the advancement of the state-of-the-art and state-of-the-practice; to mitigate this problem it has been suggested a formal definition of CS (see [109]); a standardization effort, supported by an IT standards body, would certainly be a major initiative in this context;

(2) Open-source CS detection tooling is poor, both in language coverage (Java is dominant), and in CS coverage (e.g. only a small percentage of Fowler's catalog is supported);

(3) Primary studies reporting experiments on CS often do not make the corresponding scientific workflows and datasets available, thus not allowing their "reproduction", where the goal is showing the correctness or validity of the published results;

(4) Replication of CS experiments, used to gain confidence in empirical findings, is also limited due to the effort of setting up the tooling required to running families of experiments, even when curated datasets on CS exist;

(5) Thresholds for deciding on CS occurrence are often arbitrary/unsubstantiated and not generalizable; in mitigation, we foresee the potential for the application of multi-criteria decision criteria approaches that take into account the scope and context of CS, as well as approaches that explore the power of the crowd, such as the one proposed in [113];

(6) CS studies in mobile and web environments are still scarce; due to their importance of those environments in nowadays life, we see a wide berth for CS research in those areas;

(7) CS visualization techniques seem to have great potential, especially in large systems, to help developers in deciding if they agree with a CS occurrence suggested by an existing oracle; a large research effort is required to enlarge CS visualization diversity, both in scope (single method, single class, multiple classes) and coverage, since the existing literature only tackles a small percentage of the cataloged CS.

2.7 Summary

This chapter presented the state-of-the-art techniques and tools used for code smells detection and visualization and confirms the research problems addressed by this dissertation using an SLR validated by online survey's. We confirmed that the detection of code smells is a non trivial task, and there is still a lot of work to be done in terms of: reducing the subjectivity associated with the definition and detection of code smells; increasing the diversity of detected code smells and of supported programming languages; constructing and sharing oracles and datasets to facilitate the replication of code smells detection and visualization techniques validation experiments; performing replication studies.

Part II.

CS Detection and Visualization

PART I: FUNDAMENTALS

Introduction
Chapter 1

State of the Art
Chapter 2

PART II: CODE SMELLS DETECTION AND VISUALIZATION

Crowdsmelling: The use of collective knowledge in code smells detection
Chapter 3

Smelly Maps
Chapter 4

PART III: CROWDSMELLING: A ML-BASED CROWDSOURCING APPROACH FOR CODE SMELLS DETECTION

Crowdsmelling Tool
Chapter 5

PART IV: CONCLUSION

Conclusion
Chapter 6

This part presents the first results of our *Crowdsmelling* approach for detecting and visualizing CS in a real context, and *Smelly Maps*, our proposed visualization of CS.

Chapter 3.

Crowdsmelling: The use of collective knowledge in CS detection

Contents

3.1 Introduction 55
3.2 Related Work 55
3.2.1 Crowd and collaborative-based approaches 55
3.2.2 Multiple ML models based approaches 57
3.3 Experiment Planning 58
3.3.1 Research Questions 58
3.3.2 Participants 58
3.3.3 Data 59
3.3.4 CS 61
3.3.5 Code Metrics 61
3.3.6 Machine Learning Techniques Experimented 61
3.3.7 Model Evaluation 62
3.3.8 Process 63
3.4 Results 67
3.4.1 **RQ1. What is the performance of ML techniques when trained with data from the crowd?** 67
3.4.2 **RQ2. What is the best ML model to detect each one of the three CS?** 70
3.4.3 **RQ3. Is it possible to use Collective Knowledge for CS detection?** 72
3.5 Discussion 74
3.5.1 Research Questions (RQ) 74
3.5.2 Implications and limitations of the *Crowdsmelling* Approach 76
3.5.3 Threats to validity 77
3.6 Summary 79

This chapter evaluates our approach, *Crowdsmelling*, in a real-world scenario to provide evidence that this approach is feasible for detecting CS.

3.1 Introduction

This chapter presents the results of a validation experiment for the *Crowdsmelling* approach proposed in Chapter 1.2.3. The latter is based on supervised ML techniques, where the wisdom of the crowd (of software developers) is used to collectively calibrate CS detection algorithms, thereby lessening the subjectivity issue.

In the context of three consecutive years of a Software Engineering course, a total "crowd" of around a hundred teams, with an average of three members each, classified the presence of 3 CS (*Long Method*, *God Class*, and *Feature Envy)* in Java source code. These classifications were the basis of the oracles used for training six ML algorithms. Over one hundred models were generated and evaluated to determine which ML algorithms had the best performance in detecting each CS mentioned above.

Good performances were obtained for *God Class* detection (*ROC*=0.896 for *Naive Bayes*) and *Long Method* detection (*ROC*=0.870 for *AdaBoostM1*), but much lower for *Feature Envy* (*ROC*=0.570 for *Random Forest*).

The results suggest that *Crowdsmelling* is a feasible approach for detecting CS. However, further validation experiments based on dynamic learning are required to comprehensive coverage of CS to increase external validity.

3.2 Related Work

The related work is organized in two subsections and chronologically within each one.

3.2.1 Crowd and collaborative-based approaches

Palomba et al. [99] presented LANDFILL, a Web-based platform for sharing code smell datasets, and a set of Application Program Interface (API) for programmatically accessing LANDFILL's contents. This platform was created due to the lack of publicly available oracles (sets of annotated CS). The web-based platform has a dataset of 243 instances of five types of CS (Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy) identified from 20 open-source software projects and a systematic procedure for validating code smell datasets. LANDFILL allows anyone to create, share, and improve code smell datasets.

Oliveira et al. [93] performed a controlled experiment involving 28 novice developers aimed at assessing the effectiveness of collaborative practices in the identification of CS. The authors used Pair Programming (PP) and Coding Dojo Randori (CDR), which are two increasingly adopted practices for improving the effectiveness of developers with limited or no knowledge in Software Engineering tasks, including code review tasks, and compared these two practices (PP and CDR) with solo programming in order to better distinguish their impact on the effective identification of CS. The results suggest that collaborative practices contribute to the effectiveness of identifying a wide range of CS. For nearly all types of inter-class smells, the average of CS identified by novice pairs or groups outperformed at least 40% of the corresponding average of CS identified by individuals, and collaborative practices tend to increase the success rate in identifying more complex ones. In the same year, [95] performed research based on a set of controlled experiments conducted with more than 58 novice and professional developers, with

the aim of knowing how to improve the efficiency in the collaborative identification of CS, and reached the same conclusions as to the first study.

Oliveira et al. [94] reported an industrial case study aimed at observing how 13 developers individually and collaboratively performed smell identification in five software projects from two software development organizations [94]. The results are in line with previous studies by these authors, where they suggest that collaboration contributes to improving effectiveness in identifying a wide range of CS [93, 95].

de Mello et al. [30] presented and discussed a set of context factors that may influence the effectiveness of CS identification tasks. The authors presented an initial set of practical suggestions for composing more effective teams for the identification of CS. These suggestions were, i) be sure all team professionals are aware of the code smell concepts applied in the review, ii) be sure all team professionals are aware of the relevance of identifying CS, iii) take preference to use collaboration in the reviews, iv) include professionals that had worked in the module and professionals without such experience, v) include professionals with different professionals roles.

Tahir et al. [130] presented a study where they investigated how developers discuss CS and anti-patterns over Stack Overflow to understand better their perceptions and understanding of these two concepts. In this paper, both quantitative and qualitative techniques were applied to analyze discussions containing terms associated with CS and anti-patterns. The authors reached conclusions like: i) developers widely use Stack Overflow to ask for general assessments of CS or anti-patterns, instead of asking for particular refactoring solutions, ii) developers very often ask their peers 'to smell their code' (i.e., ask whether their own code 'smells' or not), and thus, utilize Stack Overflow as an informal, crowd-based code smell/anti-pattern detector, iii) developers often discuss the downsides of implementing specific design patterns, and 'flag' them as potential anti-patterns to be avoided. Conversely, the authors also found discussions on why some anti-patterns previously considered harmful should not be flagged as anti-patterns, iv) C#, JavaScript and Java were the languages with most questions on CS and anti-patterns, constituting 59% of the total number of questions on these topics, v) Blob, Duplicated Code and *Data Class* are the most frequently discussed smells in Stack Overflow, vi) when authors analyzed temporal trends in posts on CS and anti-patterns in Stack Overflow, show that there has been a steady increase in the numbers of questions asked by developers over time.

Oliveira et al. [96] have carefully designed and conducted a controlled experiment with 34 developers. The authors exploited a particular scenario that reflects various organizations: novices and professionals inspecting systems they are unfamiliar with. They expected to minimize some critical threats to the validity of previous work. Additionally, they interviewed five project leaders aimed to understand the potential adoption of collaborative smell identification in practice. Statistical testing suggested 27% more *Precision* and 36% more *Recall* through the collaborative smell identification for both novices and professionals. The interviews performed by the authors showed that leaders would strongly adopt collaborative smell identification. However, some organization and tool constraints may limit such adoption.

Baltes and Treude [8] presented a study with similarities and differences between code clones in general and code clones on *Stack Overflow* and point to open questions that need to be addressed to be able to make data-informed decisions about how to properly handle clones on

this important platform. The results of his first preliminary investigation indicated that clones in *Stack Overflow* are common, diverse, similar to clones in regular software projects, affect the maintainability of posts, and can lead to licensing issues. The authors further point out to specific challenges, including incentives for users to clone successful answers and difficulties with bulk edits on the platform.

3.2.2 Multiple ML models based approaches

Regarding the use of the ML approach in the detection of CS, most studies only use one algorithm, being the most usual algorithm the decision trees. Therefore, we will present below the most relevant studies that use multiple ML algorithms.

Some of the most relevant studies in the area of ML were performed by Fontana et al. [6, 42]. Initially, Fontana et al. [42] outlined some common problems of code smell detectors and described the approach they were following based on ML techniques. They, them focused on four CS (*Data Class*, *Large Class*, *Feature Envy*, *Long Method*), considered 76 systems for analysis and validation, and experimented with six different ML algorithms. The results with a use of 10-fold cross-validation to assess the performance of predictive models showed that *J48*, *Random Forest*, *JRip*, and *SMO* had *Accuracy* values greater than 90% for the four CS, and on average, they had the best performances. In a following, Fontana et al. [6] performed the largest experiment so far of applying ML algorithms in this context. They experimented with 16 different machine-learning algorithms on the same four CS detected upon and 74 software systems, with 1986 manually validated code smell samples. They found that all algorithms achieved high performances in the cross-validation data set, yet the highest performances were obtained by *J48* and *Random Forest*, while support vector machines achieved the worst performance. The authors concluded that the application of ML to detect these CS could provide high *Accuracy* (>96 %), and only a hundred training examples are needed to reach such high *Accuracy*. The authors interpret the results as indicating that "using ML algorithms for code smell detection is an appropriate approach".

Di Nucci et al. [31] replicated the Fontana et al. [6] study with a different dataset configuration. The dataset contains instances of more than one type of smell, with a reduced proportion of smelly components and a smoothed boundary between the metric distribution of smelly and non-smelly components, and therefore more realistic. The results revealed that with this configuration, the ML techniques reveal critical limitations in state-of-the-art, which deserve further research. Furthermore, they concluded that when testing code smell prediction models on the revised dataset, they noticed: i) *Accuracy* of all the models is still noticeably high when compared to the results of the reference study (on average, 76% vs. 96%), ii) that performances are up to 90% less accurate in terms of *F-Measure* than those reported in the Fontana et al. study. Thus, the problem of detecting CS through the adoption of ML techniques may still be worthy of further attention, e.g., in devising proper ML-based code smell detectors and datasets for software practitioners.

To the best of our knowledge, namely obtained while performing a systematic literature review on CS detection techniques (see chapter 2.1), there is no study that uses a collective

knowledge-based approach to detect CS automatically, i.e. based on ML, with a dataset increment over 3 years. The use of groups of people in CS detection is typically used in manual detection approaches and in the construction of oracles (a tagged dataset for training detection algorithms). A distinctive feature of our approach is the use of crowds. While in related work a group of 3 to 5 people is typically used to build an oracle, we used hundreds, thus embodying a much larger large diversity of opinions.

3.3 Experiment Planning

3.3.1 Research Questions

The concept of *Crowdsmelling* – the use of collective intelligence in detecting CS – aims to mitigate the problems mentioned above of subjectivity and lack of calibration data required to obtain accurate detection model parameters by using ML techniques. We have formulated the following research questions to assess the feasibility of *Crowdsmelling*:

- **RQ1:** What is the performance of ML techniques when trained with data from the crowd and hypothetically more realistic?
- **RQ2:** What is the best ML model to detect each one of the three CS?
- **RQ3:** Is it possible to use collective knowledge for CS detection?

The goal of these RQs is to understand if our *Crowdsmelling* approach is feasible. For this, it is fundamental to understand the performance of ML techniques (RQ1), which will make our approach feasible. However, it is always important to know, in addition to performance, if that performance can optimized differently depending on the considered CS (RQ2). If it is found that there is a tendency for one algorithm to overlap with the others, in the future, we can simplify our research, focusing on fewer algorithms. This aspect will also propose the simplification of an application that automates this approach. Finally, based on this data, we intend to determine the feasibility of this approach in detecting CS (RQ3).

3.3.2 Participants

In our approach several teams use a tool, as an advisor, to detect CS and then manually confirm the detection's validity. In addition to the CS detected by the advisor tool, teams could always add other CS manually. In the end, code identification, code metrics, and classification (presence or absence of CS) were saved by creating an oracle for each code smell. This oracle was aimed at training ML algorithms for CS detection. These oracles were extended over a period of three years. In each year the oracle was enriched with data from new teams, thus increasing the variability of existing classifications, since different teams may interpret the definition of CS differently.

Our subjects were finalists (3rd year) of a B.Sc. degree in Computer Science at the ISCTE-IUL University, attending a compulsory Software Engineering course. They had similar backgrounds, as they had been trained across the same set of courses along their academic path. However, there were differences between the students, as the skills and experience in code

development were different. The knowledge about CS was acquired in the aforementioned Software Engineering course.

Table 3.1: Teams whose CS detection was included in the oracles

Year	Number of teams	Total number of elements	Average group size
2018	8	31	4
2019	51	152	3
2020	44	179	6
	103	362	

Teams had a variable size depending on the year (see Table 3.1), and the number of participants increased thought the observation period. In 2018, teams were formed, mostly with four elements each, for a total of 73 elements, but in the end, only the data from 8 teams, for a total of 31 elements, were used for the oracle. In subsection 3.3.3 we explain why the data from 11 teams were not used. In 2019 we had 51 teams, mainly made up of 3 members, with a total of 152 members. In 2020 we had 44 teams, mainly made up of 6 members, with a total of 179 members. These teams were requested to complete a CS detection assignment.

3.3.3 Data

Participants were invited to perform the detection of 3 CS (*God Class*, *Feature Envy*, *Long Method*) in a code extract (e.g., of their choice). They used *JDeodorant*[1] as an auxiliary tool in the detection. The use of tools to help detect CS in the process of creating oracles is usual. For example, in the Fontana et al. [6] study, five advisors were used, depending on the code smell that was intended to be detected. We chose *JDeodorant* because:

- Detects refactoring opportunities for the three CS used;
- Is one of the best known and used tools, as we can see in the paper by Tsantalis et al. [133];
- Integrate nicely with the *Eclipse IDE* that all subjects were used to.

To account for individual judgement in the oracle, teams could either decide to accept (true positives) or not (false positives) the tool suggestions or add additional manual detections (false negatives).

In 2018, each team chose the Java project where they wanted to do CS detection from a list of 8 open-source projects. The latter had already been used in other studies, namely in those using ML approaches mentioned in the related work section [6, 31, 42]. However, in the end, only three projects/versions were considered: *Jasml-0.10*[2], *Jgrapht-0.8.1*[3] and *Jfreechart-1.0.13*[4]. We discarded the collected data from the other projects chosen by 11 teams (42 participants) since those teams used diversified versions and, therefore, the collected metrics were not consistent

[1] https://users.encs.concordia.ca/ nikolaos/jdeodorant/
[2] http://jasml.sourceforge.net/
[3] https://jgrapht.org/
[4] https://www.jfree.org/

across versions, which would be a validity threat. We just used *Jasml-0.10* in the following two years to avoid this issue.

The results of each team's detection were saved in a file with the following fields for each of the CS *Feature Envy* and *Long Method*: *team number*, *project*, *package*, *class*, *method*, *82 metrics of code*, *is code smell*. In the case of the code smell *God Class*, as the scope is a class, the file did not have the method *field*, and 61 code metrics were collected. So, in the end, we have three files, one for each code smell.

The data obtained each year served to reinforce the calibration datasets of the ML algorithms, with the objective of improving their detection performance over time. Having several datasets allows determining the one that provides the best results for each code smell.

Table 3.2: Datasets (Oracles) and their composition

Dataset	Code smell	# Cases	True	% True	False	% False
2018	Feature Envy	10	3	30%	7	70%
2019	Feature Envy	197	110	56%	87	44%
2019+2018	Feature Envy	207	113	55%	94	45%
2020	Feature Envy	123	79	64%	44	36%
2020+2019	Feature Envy	320	189	59%	131	41%
2020+2019+2018	Feature Envy	330	192	58%	138	42%
2018	God class	22	8	36%	14	64%
2019	God class	129	74	57%	55	43%
2019+2018	God class	151	82	54%	69	46%
2020	God class	136	84	62%	52	38%
2020+2019	God class	265	158	60%	107	40%
2020+2019+2018	God class	287	166	58%	121	42%
2018	Long Method	59	24	41%	35	59%
2019	Long Method	414	180	43%	234	57%
2019+2018	Long Method	473	204	43%	269	57%
2020	Long Method	853	350	41%	503	59%
2020+2019	Long Method	1267	530	42%	737	58%
2020+2019+2018	Long Method	1326	554	42%	772	58%

In Table 3.2 we present the composition of the datasets, indicating the following elements, i) name of the dataset, ii) code smell to which the dataset refers, iii) number of cases, iv) number of true instances, v) percentage of true instances, vi) number of false instances, vii) percentage of false instances. Each dataset is identified by the year, or the years that constitute it, for example, 2019 is the dataset of the year 2019, and 2019+2020 is the dataset resulting from the aggregation of the datasets of the years 2019 and 2020. Unlike several authors, such as Fontana et al. [6], we do not normalize our datasets in size in order to balance the number of positive and negative instances. Even with the risk of getting worse results, we used the datasets with all the cases classified by the teams. Thus, we believe that we are reproducing the reality of the teams' thinking about CS. The size of the datasets varies widely depending on the type of code smell. Since the datasets of the code smell *Feature Envy* are very small, i.e., for a code smell in the scope of the method, they do not have a large enough variance of cases, it was not possible to obtain good results. Even so, we intend to use all the datasets, as they represent the obtained

reality and serve as a basis for a future amplification and evolution of the crowd's study in CS detection.

The 18 datasets are available on GitHub[5] and Zenodo [112].

3.3.4 CS

In this study, we considered three different types of CS defined by Fowler et al. [43]:

- *Feature Envy*. When a method is more interested in members of other classes than its own, it is a clear sign that it is in the wrong class;
- *Long Method*. They usually are very large and complex and, therefore, difficult to understand, extend and modify. It is very likely that they are implementing more than one functionality, therefore hurting one of the principles of a good Object Oriented design, the Single Responsibility Principle (SRP);
- *God Class*. This smell characterises classes having a large size, poor cohesion, and several dependencies with other data classes of the system. Class that has many responsibilities and therefore contains many variables and methods. The same SRP also applies in this case;

The choice of these three codes smells was due to the fact that, according to the Systematic Literature Review we conducted, they are the three most detected CS (see section 2.4.5). Therefore, it is easier for teams to obtain documentation and understand these three CS for better detection.

3.3.5 Code Metrics

In this study, we used the same metrics that were used in the study of Fontana et al. [6], since the metrics values for 74 java projects are publicly available[6].

The metrics extracted from the software, which constitute the independent variables in the ML algorithms, are at class, method, package, and project level. For *God Class*, we used a set of 61 metrics, and for the other two CS, *Feature Envy* and *Long Method*, we used a set of 82 metrics. Those metrics are listed in table B.1 and their definition and description can be found in the study of Fontana et al. [6].

3.3.6 Machine Learning Techniques Experimented

The tool used in this experiment to train and evaluate ML algorithms was *Weka* (open-source software from Waikato University) [51], and the following algorithms available in Weka were used:

- *J48* [106] is an implementation of the *C4.5* decision tree and its three types of pruning techniques: pruned, unpruned, and reduced error pruning;

[5] https://github.com/dataset-cs-surveys/Crowdsmelling

[6] https://essere.disco.unimib.it/machine-learning-for-code-smell-detection/

- *Random Forest* [14] consists of a large number of individual decision trees, a forest of random trees, that operate as an ensemble;
- *AdaBoostM1* [44] Boosting works by repeatedly running a given weak learning algorithm on various distributions over the training data and then combining the classifiers produced by the weak learner into a single composite classifier. *Weka* uses the *Adaboost M1* method;
- *SMO* [104] is a Sequential Minimal Optimization algorithm widely used for training support vector machines. We use the Polynomial kernel;
- *Multilayer Perceptron* [115] is a classifier that uses backpropagation to learn a multi-layer perceptron to classify instances;
- *Naïve Bayes* [56] is a probabilistic model based on the Bayes theorem.

Experiments were performed to evaluate the ML algorithms' performance with their default parameters for each type of code smell. Also, no feature selection technique was used.

3.3.7 Model Evaluation

To assess the capabilities of the ML model, we adopted 10-Fold Cross-Validation [128]. This methodology randomly partitions the data into 10 folds of equal size, applying a stratified sampling (e.g., each fold has the same proportion of code smell instances). A single fold is used as a test set, while the remaining ones are used as a training set. The process was repeated ten times, using a different fold each time as a test set. The result of the process described above consisted of a confusion matrix for each code smell type, and each model [103].

Several evaluation metrics can be used to assess model quality in terms of false positives/negatives (FP/FN) and true classifications (TP/TN). However, commonly used measures, such as ***Accuracy, Precision, Recall,* and *F-Measure*** as defined in equations 3.1, 3.2, 3.3, 3.4, do not perform very well in the case of an imbalanced dataset, or they require the use of a minimum probability threshold to provide a definitive answer for predictions. For these reasons, we used the ***ROC***[7], which is a threshold invariant measurement [18]. Nevertheless, for general convenience, we provide all the evaluation metrics in the results tables.

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{3.1}$$

$$Precision = \frac{TP}{TP + FP} \tag{3.2}$$

$$Recall = \frac{TP}{TP + FN} \tag{3.3}$$

$$F\text{-}Measure = 2 * \frac{Recall * Precision}{Recall + Precision} \tag{3.4}$$

[7] Receiver Operating Characteristic (***ROC***) is a curve that plots the true positive rates against the false positive rates for all possible thresholds between 0 and 1.

ROC gives us a 2-D curve, which passes through (0, 0) and (1, 1). The best possible model would have the curve close to y = 1, with an area under the curve (**AUC**) close to 1.0. **AUC** always yields an area of 0.5 under random guessing. This enables comparing a given model against random prediction without worrying about arbitrary thresholds or the proportion of subjects on each class to predict [107].

3.3.8 Process

This subsection describes the three stages that constitute the process adopted in this exploratory study.

3.3.8.1 Stage 1: Developer - Code smell classification

All Java developers used the Eclipse IDE with the *JDeodorant* plug-in installed.

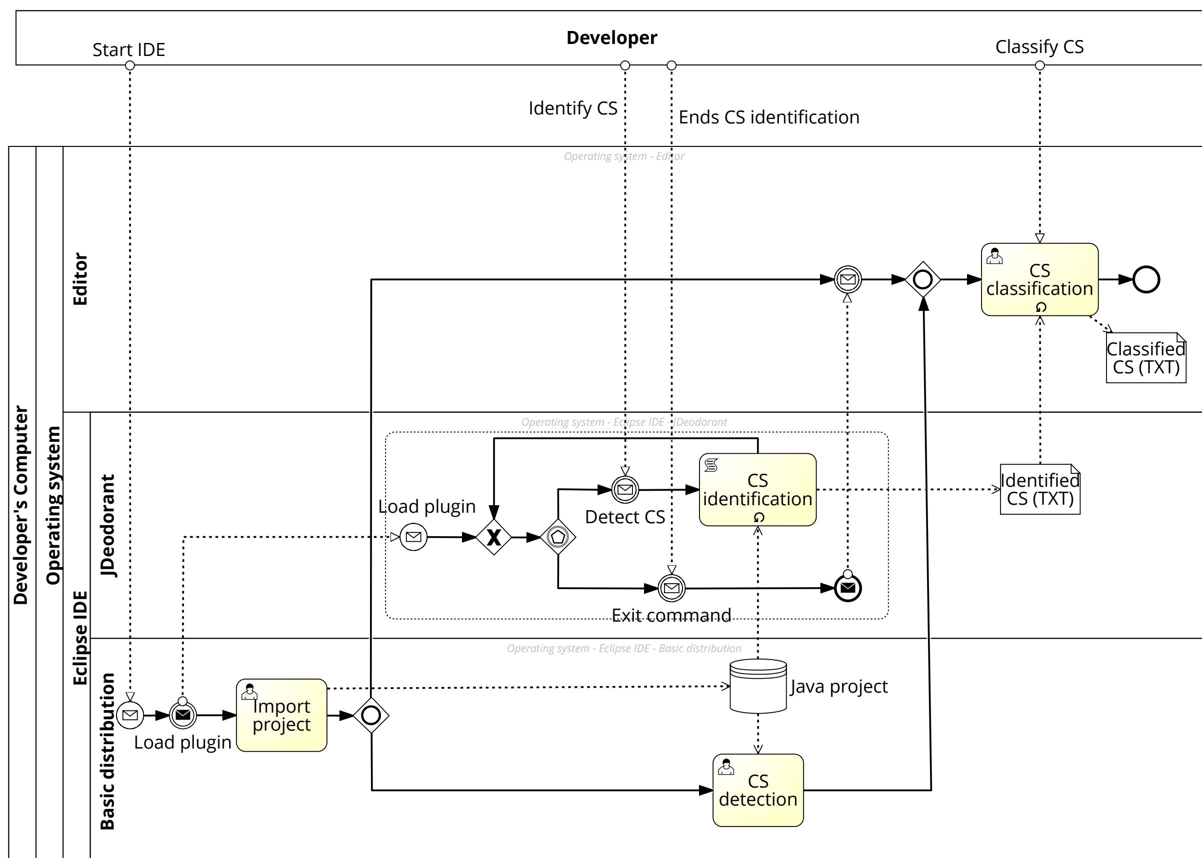


Figure 3.1: Process of CS classification by the developer

Figure 3.1 shows the CS classification process by the programmer, where we can see that after importing the Java project, the participants were invited to perform the detection of 3 Code smells (*Long Method*, *God Class*, *Feature Envy*). In 2018, detections of CS were performed in the 3 Java projects as follows:

a) *Long Method*, five teams detected this smell in jasml-0.10, two teams detected it in jfreechart-1.0.13, and all detections performed in jgrapht-0.8. One was not used for the reasons described in subsection 3.3.3;

b) *God Class*, four teams detected this smell in jasml-0.10, two teams detected it in jfreechart-1.0.13, and one team detected it in jgrapht-0.8.1;

c) *Feature Envy*, only the detections made by four teams in jasml-0.10 were used, all other detections were discarded for the reasons already mentioned.

In the following two years, all teams detected all three CS in jasml-0.10.

In this detection, the participants could use *JDeodorant* as an auxiliary tool to detect smells, i.e., they first used *JDeodorant* as an advisor and then manually validated the result of the detection by agreeing or not, with the detected CS. Furthermore, *JDeodorant* detects refactoring opportunities (refactoring is a controlled technique for improving the design of an existing code base [43]), consequently, when *JDeodorant* detects a refactoring opportunity, it is detecting a code smell candidate. Finally, the use of *JDeodorant* also had the advantage that participants could export the CS identified by this tool to a text file, where they later registered their agreement or not with this identification, i.e., they performed the final classification.

Regardless of the use of *JDeodorant*, all participants could identify the CS directly in the Java project code (using the code metrics) and record their occurrence or not in a text file. In this case, the participants wrote in the text file the name of the class or method and if there existed or not a code smell. The percentage of teams that performed CS detection without the help of the *JDeodorant* advisor was 7%. Although the work of detecting CS without the use of the advisor is higher, we found that, on average, the teams that did not use the advisor detected 30% more *Long Methods* and 20% more *God Class*. Regarding *Feature Envy*, the detection was on average 16% less than the teams that used *JDeodorant*.

With the use of *JDeodorant*, as an advisor, in detecting smells, there is a risk that teams will only classify CS resulting from advisor detection, in our case, CS candidates detected by *JDeodorant*. The teams were asked to classify all classes and methods in a project package to mitigate this risk, thus extending the classification to cases not detected by *JDeodorant*. Another factor that minimizes this risk is the fact that *JDeodorant* identifies refactoring opportunities in code that is clearly not code smell, but the code can still be improved. This fact causes *JDeodorant's* detection to result in a larger percentage of false positives and consequently a larger disagreement between the teams' classification and *JDeodorant's* identification. In the detection of the *Long Method*, the degree of disagreement with *JDeodorant* in the year 2018 was 66% (highest disagreement), and in the year 2019, it was 49% (lowest disagreement), being in total for the three years 54%. For *God Class*, the disagreement with *JDeodorant* for the three years was 47% and varied from 68% in the year 2018 to 40% in 2020. In *Feature Envy*, the disagreement ranged from 70% in 2018 to 34% in 2020, being in the three years 45%.

Regarding the code classified by the teams, methods, and classes of the applications, we found that the majority was classified by more than one team. In the first year, 2018, due to the diversity of Java projects used, there was a greater dispersion of the code classified, with most classes (75%) and methods (76%) classified by only one team. The most extreme cases were a class classified by 4 teams and a method classified by 6 teams. The next two years saw a reversal, with most classes and methods being classified by more than one team. Regarding classes, 60% in the year 2019 and 75% in the year 2020 were classified by more than one team, with the most extreme case was a class being classified by 43 teams. Regarding methods, 85%

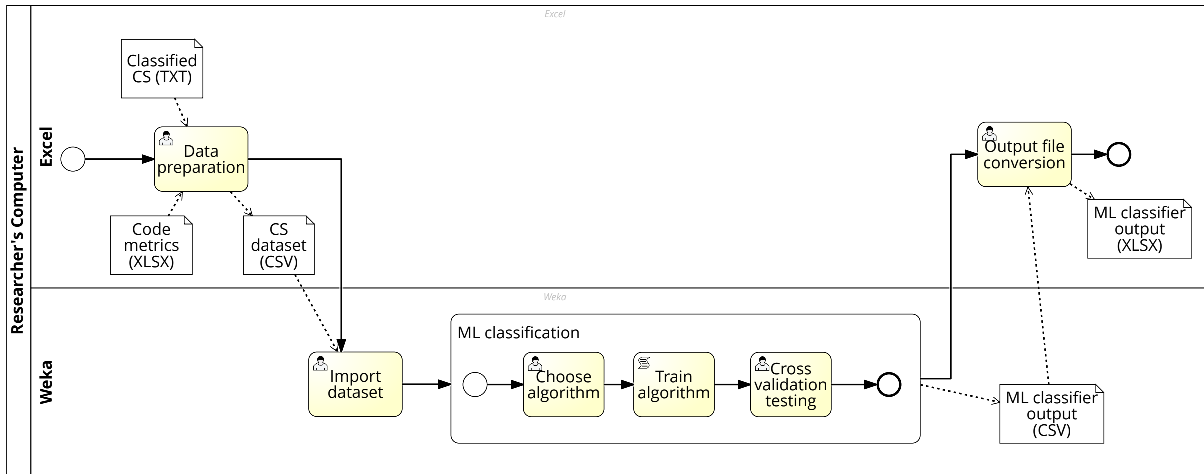


Figure 3.2: Process of creation of the datasets and evaluation of the ML techniques by the researcher

and 60% were classified by more than one team in 2020 and 2019, respectively, with the most extreme case was a method being classified by 44 teams.

The time given to the teams to classify the three CS was three weeks, and no indication was given on how they should work as a team, that is, how they should divide the CS classification among the various team members. Hence, based on the data obtained from the experiment, we cannot identify precisely which members performed a specific code analysis. However, we were able to identify which CS were analyzed. For example, according to data available in GitHub[8] and Zenodo [112], it is possible to identify that in the 2020 *Long Method* dataset, the *private void consumeDigits()* method was classified by 37 teams by applying a filter to the method field. Furthermore, we have made available on GitHub and Zenodo the file *code-classification-statistics.csv* with a set of statistics about the percentages of teams that classified the methods and classes. We also found that the teams divided the classification of the three CS among their members, for example, when the team had six members, they created groups of two members, and each group classified one code smell in the code. In this way, the teams increased the reliability of the classification since two team members classified the code.

As a result of this stage, all teams produced three files - one for each code smell - with the classification of a set of methods and classes of the Java project, i.e., with the record of the existence or not of CS in those classes or methods. This stage was performed over three years, 2018, 2019, and 2020.

3.3.8.2 Stage 2: Researcher - Evaluation of ML models

After collecting data in three years, we proceeded to the second phase, which aimed to produce the datasets for the three CS and evaluate the different ML techniques. In figure 3.2 is represented the whole process of this second stage.

The first task to be performed by the researcher was the creation of the datasets described in section 3.3.3.

[8] https://github.com/dataset-cs-surveys/Crowdsmelling

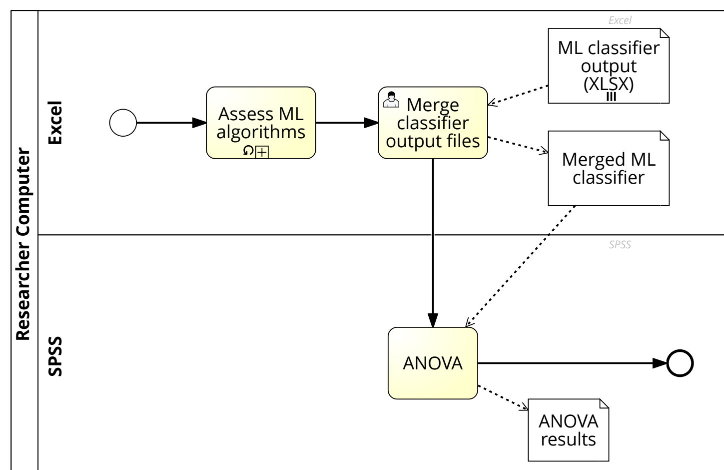


Figure 3.3: Process of testing the variance between ML models

The creation of the datasets was done by joining all the text files with the classifications of a code smell produced by the teams of each year in a single *Excel* file. Then, to this *Excel* file were added the code metrics for the methods or classes (see section 3.3.5), depending on the scope of the code smell to which the dataset belongs. Thus, in a first step, datasets were created - usually called oracles - with the data for each year, for each of the three CS, for a total of 6 datasets. These datasets were given the name of the year when they were collected, i.e., 2018, 2019, and 2020. In a second step, we aggregated the dataset of the year with those of previous years to make the dataset larger, increasing the number of instances. In the end, we created six datasets for each code smell, totalling 18 datasets (see table 3.2).

After creating the datasets, we proceeded to the creation and evaluation of the ML models using *Weka* (open-source software from Waikato University) [51]. To import datasets into *Weka*, we convert the datasets files from *Excel* XLSX to Comma Separated Values (CSV). With *Weka* we trained the six algorithms described in section 3.3.6, with each of the 18 datasets, and evaluated the model produced using the 10-Fold Cross-Validation methodology. In the end, 36 ML models were created for each code smell, with a total of 108 models for the three CS. Finally, all the metrics (*Accuracy*, *Precision*, *Recall*, *F-Measure*, and *ROC*) resulting from the evaluation of each model were saved in the "ML classifier output"file (see section 3.3.7).

3.3.8.3 Stage 3: Researcher - Model variance test

To check if there were significant differences among the classifications presented by the different models, we applied a one-way ANalysis Of VAriance (ANOVA) (see figure 3.3).

We used the *ROC* value to test the variance among the ML models. Thus, the first step was to produce a data file, for each code smell, with the identification of the ML models and the respective *ROC*. This file was created aggregating the results of the evaluations of all models produced by Weka per code smell.

To analyze if there were differences between the classifications of the ML models for each code smell, we performed an analysis of variance using a one-way analysis of variance (ANOVA) test using the IBM SPSS Statistics 27 software.

3.4 Results

In this section, we present the experiment results with respect to our research questions.

3.4.1 RQ1. What is the performance of ML techniques when trained with data from the crowd?

In this **RQ** we will evaluate the performance of the 36 models for each code smell in a total of 108 models. These models resulted from the training of the six ML algorithms (*J48*, *Random Forest*, *AdaBoostM1*, *SMO*, *Multilayer Perceptron*, *Naïve Bayes*), described in section 3.3.6, by the datasets presented in table 3.2. These algorithms were trained with the various datasets resulting from the crowd and, as explained in 3.3.3, we expect these datasets to be realistic because, to represent as faithfully as possible what the detection teams think about the CS.

We have chosen to use the *ROC* as the primary metric from the various existing metrics for evaluating ML models, but we also use *Accuracy*, *Precision*, *Recall*, and *F-Measure*. For testing, we used the 10-Fold Cross-Validation as justified in 3.3.7.

3.4.1.1 Performance of ML techniques for the code smell *Long Method*

Starting by analyzing the ML techniques for the *Long Method* data, described in Table 3.3, we observed that the *Random Forest* and *AdaBoostM1* algorithms obtained the best results. The best result with a *ROC* of 0.870 was obtained by *AdaBoostM1* when trained by the 2020 dataset, followed by the *Random Forest* with a *ROC* of 0.869 for the same dataset. For the 2018 dataset, the best result was also that of *AdaBoostM1*. However, the most uniform algorithm was *Random Forest*, with the best results in 4 of the six datasets (2020+2019+2018, 2020+2019, 2019+2018, 2019), and for the 2020 dataset, the difference for *AdaBoostM1* is insignificant (0.001). The *Multilayer Perceptron* and *J48* algorithms were two other algorithms to present *ROC* results above 0.800. Especially the *Multilayer Perceptron* algorithm, which for the datasets of the year 2020 presented a *ROC* between 0.868 and 0.822.

Table 3.3: *Long Method*: *ROC* Area results for the ML algorithms trained by the 3 years datasets

Algorithm \ year / dataset	2020			2019		2018
	2020+2019+2018	2020+2019	2020	2019+2018	2019	2018
J48	0.792	0.801	0.832	0.677	0.678	0.617
Random Forest	0.828	0.828	0.869	0.684	0.679	0.671
AdaBoostM1	0.807	0.818	0.870	0.665	0.673	0.707
SMO	0.753	0.753	0.803	0.634	0.649	0.524
Multilayer Perceptron	0.822	0.822	0.868	0.683	0.667	0.604
Naïve Bayes	0.736	0.742	0.783	0.584	0.614	0.471

The worst results were obtained by the *Naïve Bayes* algorithm with *ROC* between 0.783 and 0.471. The second worst algorithm was *SMO*, with *ROC* results between 0.803 and 0.524.

In table 3.3, we can still observe that the best results were obtained when the algorithms were trained with the 2020 datasets, with *ROC* of 0.870 for the 2020 dataset and *ROC* of 0.828

for the datasets 2020+2019+2018 and 2020+2019. In opposition is 2019, with the worst results, *ROC* of 0.684 and 0.679 for the datasets 2019+2018, 2019, respectively.

3.4.1.2 Performance of ML techniques for the code smell *God Class*

Table 3.4 shows the results of the ML techniques for the *God Class* data. The best result was obtained by the *Naïve Bayes* algorithm, when trained by the 2020 dataset, with the *ROC* value of 0.896. The algorithms that obtained the best performances were *Naïve Bayes* and *Multilayer Perceptron*, with the best result in 3 of the datasets, each one. *Naïve Bayes* obtained the best results for the datasets 2020, 2020+2019, 2019, with *ROC* values of 0.896, 0.859, and 0.804, respectively. Also, with the best result in 3 datasets (2020+2019+2018, 2019+2018, 2018), the *Multilayer Perceptron* algorithm presented *ROC* values between 0.768 and 0.885. The *Random Forest* and *AdaBoostM1* algorithms presented their best *ROC* values of 0.893 and 0.876, respectively, for the 2020 dataset.

Table 3.4: *God Class*: *ROC* Area results for the ML algorithms trained by the 3 years datasets

year / dataset / Algorithm	2020			2019		2018
	2020+2019+2018	2020+2019	2020	2019+2018	2019	2018
J48	0.763	0.759	0.791	0.693	0.725	0.692
Random Forest	0.853	0.850	0.893	0.781	0.802	0.491
AdaBoostM1	0.854	0.857	0.876	0.771	0.793	0.571
SMO	0.815	0.800	0.857	0.716	0.751	0.741
Multilayer Perceptron	0.880	0.853	0.885	0.805	0.797	0.768
Naïve Bayes	0.731	0.859	0.896	0.669	0.804	0.651

The worst results were presented by *J48* and *SMO*, with their best *ROC* values for the dataset 2020 of 0.759 and 0.857, respectively.

Regarding the datasets that presented the best results were those of the year 2020, with the dataset only with data of the year 2020 being the best (dataset 2020) with *ROC* values between 0.896 and 0.791. The dataset with the worst results was 2018, with a *ROC* between 0.491 and .0768.

3.4.1.3 Performance of ML techniques for the code smell *Feature Envy*

The *ROC* results for the ML algorithms trained by the 3-year datasets for the code smell *Feature Envy* are presented in table 3.5. *Feature Envy* detection results are low, with the *Random Forest* algorithm having the best *ROC* value of 0.570 when trained by dataset 2019. As already explained in point 3.3.3, the datasets for *Feature Envy* are very small, considering the variance of cases. However, we are convinced that the results will be better when we have bigger datasets. The ML algorithms showed better results when trained with the datasets of the year 2019, with *ROC* values between 0.570 and 0.508.

Table 3.5: *Feature Envy*: *ROC* Area results for the ML algorithms trained by the 3 years datasets

year	2020			2019		2018
Algorithm \ dataset	2020+2019+2018	2020+2019	2020	2019+2018	2019	2018
J48	0.518	0.484	0.467	0.552	0.563	0
Random Forest	0.539	0.494	0.486	0.542	0.570	0
AdaBoostM1	0.498	0.437	0.468	0.554	0.548	0
SMO	0.520	0.491	0.500	0.551	0.508	0
Multilayer Perceptron	0.533	0.498	0.536	0.548	0.544	0
Naïve Bayes	0.524	0.519	0.482	0.548	0.547	0

3.4.1.4 The one-way analysis of variance (ANOVA)

To determine if there were significant differences among the performance of ML techniques when trained with data from the crowd, a One-way ANOVA was conducted to compare the effect of ML techniques on the *ROC*. Before performing the ANOVA, we checked all the assumptions for its application, namely, the inexistence of outliers, the normality of the distribution (Shapiro-Wilk test), and the homogeneity of variances (Levene's test). All assumptions were fulfilled, and the following results were obtained:

(i) For the code smell *Long Method*, an analysis of variance showed that the effect of the performance of ML techniques on *ROC* value was not significant, F(5,30)=1.096, p=.383.

(ii) For the code smell *God Class*, an analysis of variance showed that the effect of the performance of ML techniques on *ROC* value was not significant, F(5,30)=.655, p=.660.

(iii) For the code smell *Feature Envy*, an analysis of variance showed that the effect of the performance of ML techniques on *ROC* value was not significant, F(5,24)=.585, p=.712.

The results of the variance tests showed there was no statistically significant difference between the performance of the six ML models when trained with data from the crowd.

3.4.1.5 Summary of RQ1 results

For the code smell Long Method the best result with a *ROC* of 0.870 was obtained by *AdaBoostM1* when trained by the 2020 dataset, followed by the *Random Forest* with a *ROC* of 0.869 for the same dataset. The best result was obtained for the code smell *God Class* by the *Naïve Bayes* algorithm, when trained by the 2020 dataset, with the *ROC* value of 0.896. On the other hand, *Feature Envy* detection results are low, with the *Random Forest* algorithm having the best *ROC* value of 0.570 when trained by dataset 2019.

The results of the variance tests (performed through One-way ANOVA) showed there was no statistically significant difference between the performance of the six ML models when trained with data from the crowd and therefore more realistic.

3.4.2 RQ2. What is the best ML model to detect each one of the three CS?

In this RQ, we wanted to know which is the best model to detect each CS. To do so, we analyzed the various metrics that evaluated the performance of CS prediction models in detecting each of the three CS. Of course, the best model will vary with the metric we choose to analyze the model performance (*Accuracy*, *Precision*, *Recall*, *F-Measure*, *ROC*), but for the reasons described in 3.3.7 we will use *ROC* as the main metric.

Tables 3.6, 3.7, and 3.8 present the performance of the prediction models for the 3 CS, where the best values for each of the evaluation metrics are marked.

3.4.2.1 Best ML model for the code smell *Long Method*

For the code smell *Long Method*, the model best performs its detection is *AdaBoostM1*, presenting the best values for all evaluation metrics. As we can see in the table 3.6, *AdaBoostM1* obtained a *ROC* value of 0.870, an *Accuracy* of 81.36%, a *Precision* of 82.90%, a *Recall* of 81.40%, and an *F-Measure* of 81.50%. However, two more models present an almost equal *ROC*, *Random Forest*, and *Multilayer Perceptron*, with *ROC* values of 0.869 and 0.868, respectively.

Except for *Naïve Bayes*, all the other five models have values higher than 0.803 for *ROC* and values higher than 80.00% for *F-Measure*, *Precision*, and *Recall* in the detection of *Long Method*.

3.4.2.2 Best ML model for the code smell *God Class*

Table 3.7 presents the results of *God Class* detection using the 10-Fold Cross-Validation technique and where the best values are marked. As we can see in table 3.7, the model that presents the best value for the *ROC* is Naive Bayes with a value of 0.896. For the remaining four evaluation metrics, the *Random Forest* model presents the same values as the Naive Bayes. Thus, the Naive Bayes and *Random Forest* models present an *Accuracy* value of 88.97%, a *Precision* value of 89.70%, a *Recall* value of 89.00%, and an *F-Measure* value of 88.70%.

When we evaluated the models by the *ROC* value, we verified that, except for the *J48* model, all the other five models had values higher than 0.857. For the remaining evaluation metrics, all six models have: a) *Accuracy* values higher or equal to 87.50%, b) *Precision* values higher or equal to 87.80%, c) *Recall* values higher or equal to 87.50%, and d) *F-Measure* values higher or equal to 87.20%.

When we compare the results of the code smell *God Class* detection with those of the *Long Method*, we verify that the results of the *God Class* are better.

3.4.2.3 Best ML model for the code smell *Feature Envy*

Regarding the code smell *Feature Envy*, we present in table 3.8 the results of the evaluation of the different models. For the 2018 dataset of *Feature Envy*, it was not possible to obtain *Precision* and consequently *F-Measure* since all the instances classified as TRUE were poorly classified, i.e., all the instances were classified as FALSE. For the 2020 dataset, we also did not obtain *Precision* and *F-Measure* because all the instances classified as FALSE were badly classified, i. e., all the models, created from this dataset to classify the *Future Envy*, classified all its instances

Table 3.6: *Long Method*: Performance of the code smell prediction models

Dataset	Classifier	Accuracy	TP Rate	FP Rate	Precision	Recall	*F-Measure*	ROC Area
2018	J48	61.02%	61.00%	37.20%	63.40%	61.00%	61.30%	0.617
2018	Random Forest	61.02%	61.00%	45.10%	60.00%	61.00%	60.00%	0.671
2018	AdaBoostM1	67.80%	67.80%	36.50%	67.30%	67.80%	67.40%	0.707
2018	SMO	55.93%	55.90%	51.20%	54.30%	55.90%	54.50%	0.524
2018	Multilayer Perceptron	57.63%	57.60%	46.10%	57.40%	57.60%	57.50%	0.604
2018	Naïve Bayes	61.02%	61.00%	47.70%	59.40%	61.00%	58.70%	0.471
2019	J48	64.73%	64.70%	34.10%	66.10%	64.70%	64.90%	0.678
2019	Random Forest	66.18%	66.20%	34.50%	66.40%	66.20%	66.30%	0.679
2019	AdaBoostM1	66.67%	66.70%	29.60%	70.70%	66.70%	66.30%	0.673
2019	SMO	65.46%	65.50%	35.70%	65.50%	65.50%	65.50%	0.649
2019	Multilayer Perceptron	63.29%	63.30%	38.60%	63.10%	63.30%	63.10%	0.667
2019	Naïve Bayes	61.11%	61.10%	40.80%	60.90%	61.10%	61.00%	0.614
2019+2018	J48	65.75%	65.80%	33.20%	67.00%	65.80%	65.90%	0.677
2019+2018	Random Forest	65.33%	65.30%	35.30%	65.60%	65.30%	65.40%	0.684
2019+2018	AdaBoostM1	66.60%	66.60%	29.10%	71.40%	66.60%	66.20%	0.665
2019+2018	SMO	63.85%	63.80%	37.10%	64.00%	63.80%	63.90%	0.634
2019+2018	Multilayer Perceptron	63.00%	63.00%	39.40%	62.70%	63.00%	62.80%	0.683
2019+2018	Naïve Bayes	58.35%	58.40%	43.70%	58.20%	58.40%	58.30%	0.584
2020	J48	79.95%	80.00%	20.30%	80.30%	80.00%	80.00%	0.832
2020	Random Forest	80.66%	80.70%	20.70%	80.60%	80.70%	80.70%	0.869
2020	AdaBoostM1	81.36%	81.40%	16.70%	82.90%	81.40%	81.50%	0.870
2020	SMO	80.77%	80.80%	20.20%	80.90%	80.80%	80.80%	0.803
2020	Multilayer Perceptron	80.07%	80.10%	21.50%	80.00%	80.10%	80.00%	0.868
2020	Naïve Bayes	73.39%	73.40%	33.00%	73.70%	73.40%	72.30%	0.783
2020+2019	J48	76.32%	76.30%	22.10%	77.80%	76.30%	76.50%	0.801
2020+2019	Random Forest	77.19%	77.20%	22.60%	77.70%	77.20%	77.30%	0.828
2020+2019	AdaBoostM1	76.80%	76.80%	20.30%	79.40%	76.80%	76.90%	0.818
2020+2019	SMO	75.53%	75.50%	25.00%	75.80%	75.50%	75.60%	0.753
2020+2019	Multilayer Perceptron	75.85%	75.80%	24.60%	76.10%	75.80%	75.90%	0.822
2020+2019	Naïve Bayes	68.43%	68.40%	35.70%	68.00%	68.40%	67.90%	0.742
2020+2019+2018	J48	76.40%	76.40%	22.70%	77.40%	76.40%	76.50%	0.792
2020+2019+2018	Random Forest	76.77%	76.80%	22.70%	77.50%	76.80%	76.90%	0.828
2020+2019+2018	AdaBoostM1	76.40%	76.40%	20.50%	79.30%	76.40%	76.50%	0.807
2020+2019+2018	SMO	75.19%	75.20%	24.60%	75.80%	75.20%	75.30%	0.753
2020+2019+2018	Multilayer Perceptron	76.92%	76.90%	22.50%	77.70%	76.90%	77.10%	0.822
2020+2019+2018	Naïve Bayes	68.18%	68.20%	35.70%	67.80%	68.20%	67.70%	0.736

as TRUE. For this reason, we will not consider in the response to the RQ the models resulting from the training by these two datasets.

When we evaluate the models by the *ROC* metric, we find that the best model is the *Random Forest* with a *ROC* of 0.570. However, if we compare the various evaluation metrics, we find that all the other evaluation metrics have better values than the *ROC* metric. The best performance in the detection of *Feature Envy* is obtained by the *Naive Bayes* model for *Precision* with a value of 61.40%. The *Random Forest* model also obtains the best *Accuracy* with 59.69% and *Recall* with a value of 59.70%.

When we compare the results of the models for the detection of the three smells, we verify that the *Feature Envy* detection models obtain the worst results.

3.4.2.4 Summary of RQ2 results

For the code smell *Long Method*, the model that best performs its detection is *AdaBoostM1*, presenting the best values for all evaluation metrics. For the *God Class*, the model that presents the best value for the *ROC* is Naive Bayes, with a value of 0.896. For *Feature Envy*, when we

Table 3.7: *God Class*: Performance of the code smell prediction models

Dataset	Classifier	Accuracy	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area
2018	J48	81.82%	81.80%	26.50%	82.00%	81.80%	81.10%	0.692
2018	Random Forest	63.64%	63.60%	47.60%	61.90%	63.60%	62.30%	0.491
2018	AdaBoostM1	68.18%	68.20%	39.60%	67.40%	68.20%	67.70%	0.571
2018	SMO	77.27%	77.30%	29.10%	76.90%	77.30%	76.90%	0.741
2018	Multilayer Perceptron	72.73%	72.70%	31.70%	72.70%	72.70%	72.70%	0.768
2018	Naïve Bayes	68.18%	68.20%	45.00%	66.70%	68.20%	66.10%	0.651
2019	J48	72.87%	72.90%	29.50%	72.70%	72.90%	72.70%	0.725
2019	Random Forest	73.64%	73.60%	28.50%	73.50%	73.60%	73.50%	0.802
2019	AdaBoostM1	72.87%	72.90%	29.50%	72.70%	72.90%	72.70%	0.793
2019	SMO	76.74%	76.70%	26.60%	76.90%	76.70%	76.30%	0.751
2019	Multilayer Perceptron	75.97%	76.00%	27.70%	76.10%	76.00%	75.50%	0.797
2019	Naïve Bayes	76.74%	76.70%	26.60%	76.90%	76.70%	76.30%	0.804
2019+2018	J48	70.86%	70.90%	30.00%	70.80%	70.90%	70.80%	0.693
2019+2018	Random Forest	67.55%	67.50%	32.40%	67.80%	67.50%	67.60%	0.781
2019+2018	AdaBoostM1	69.54%	69.50%	30.90%	69.50%	69.50%	69.50%	0.771
2019+2018	SMO	72.19%	72.20%	28.90%	72.10%	72.20%	72.00%	0.716
2019+2018	Multilayer Perceptron	71.52%	71.50%	29.00%	71.50%	71.50%	71.50%	0.805
2019+2018	Naïve Bayes	74.83%	74.80%	26.50%	74.90%	74.80%	74.60%	0.669
2020	J48	87.50%	87.50%	17.30%	87.80%	87.50%	87.20%	0.791
2020	Random Forest	88.97%	89.00%	16.40%	89.70%	89.00%	88.70%	0.893
2020	AdaBoostM1	88.24%	88.20%	16.80%	88.70%	88.20%	87.90%	0.876
2020	SMO	88.24%	88.20%	16.80%	88.70%	88.20%	87.90%	0.857
2020	Multilayer Perceptron	88.24%	88.20%	16.80%	88.70%	88.20%	87.90%	0.885
2020	Naïve Bayes	88.97%	89.00%	16.40%	89.70%	89.00%	88.70%	0.896
2020+2019	J48	82.64%	82.60%	21.70%	82.90%	82.60%	82.30%	0.759
2020+2019	Random Forest	83.02%	83.00%	21.50%	83.40%	83.00%	82.60%	0.850
2020+2019	AdaBoostM1	82.64%	82.60%	21.70%	82.90%	82.60%	82.30%	0.857
2020+2019	SMO	82.26%	82.30%	22.30%	82.60%	82.30%	81.90%	0.800
2020+2019	Multilayer Perceptron	82.26%	82.30%	22.30%	82.60%	82.30%	81.90%	0.853
2020+2019	Naïve Bayes	83.02%	83.00%	21.50%	83.40%	83.00%	82.60%	0.859
2020+2019+2018	J48	81.88%	81.90%	21.70%	82.30%	81.90%	81.50%	0.763
2020+2019+2018	Random Forest	81.53%	81.50%	22.00%	81.90%	81.50%	81.20%	0.853
2020+2019+2018	AdaBoostM1	80.84%	80.80%	22.70%	81.20%	80.80%	80.50%	0.854
2020+2019+2018	SMO	83.28%	83.30%	20.30%	83.80%	83.30%	82.90%	0.815
2020+2019+2018	Multilayer Perceptron	82.23%	82.20%	20.10%	82.20%	82.20%	82.10%	0.880
2020+2019+2018	Naïve Bayes	81.88%	81.90%	21.30%	82.10%	81.90%	81.60%	0.731

evaluate the models by the *ROC* metric, we find that the best model is *Random Forest*. However, for this code smell, the best performance is obtained by the Naive Bayes model, for the *Precision* metric, with a value of 61.40%. When we compare the results of the models for the detection of the three smells, we verify that the worst results are obtained by the *Feature Envy* detection models and the best results by *God Class*.

3.4.3 RQ3. Is it possible to use Collective Knowledge for CS detection?

Several studies present CS detection results through ML techniques with *Accuracy*, *Precision*, *Recall*, and *F-Measure*, close to 100%. However, these studies use very treated datasets to obtain good results, making the datasets unrealistic. A proof of this is the replication of one of the most important studies on CS detection using ML techniques by Di Nucci et al. [31], where more realistic datasets were used in this replication. The results of this replication showed that the *Accuracy* value, on average, decreased from 96% to 76%, but the *F-measure* presented results 90% lower than in the reference work. When we compare our results with the Di Nucci et al. [31] study, we find that the results are similar in some metrics and better in others.

Table 3.8: *Feature Envy*: Performance of the code smell prediction models

Dataset	Classifier	Accuracy	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area
2018	J48	70.00%	70.00%	70.00%	-	70.00%	-	0.000
2018	Random Forest	70.00%	70.00%	70.00%	-	70.00%	-	0.000
2018	AdaBoostM1	70.00%	70.00%	70.00%	-	70.00%	-	0.000
2018	SMO	70.00%	70.00%	70.00%	-	70.00%	-	0.000
2018	Multilayer Perceptron	70.00%	70.00%	70.00%	-	70.00%	-	0.000
2018	Naïve Bayes	30.00%	30.00%	87.10%	35.00%	30.00%	32.30%	0.000
2019	J48	56.85%	56.90%	46.10%	56.20%	56.90%	56.20%	0.563
2019	Random Forest	58.38%	58.40%	44.70%	57.80%	58.40%	57.70%	0.570
2019	AdaBoostM1	54.82%	54.80%	51.40%	52.90%	54.80%	51.40%	0.548
2019	SMO	52.79%	52.80%	51.30%	51.50%	52.80%	51.40%	0.508
2019	Multilayer Perceptron	51.78%	51.80%	52.30%	50.40%	51.80%	50.40%	0.544
2019	Naïve Bayes	52.28%	52.30%	45.40%	54.30%	52.30%	52.00%	0.547
2019+2018	J48	57.97%	58.00%	42.80%	57.90%	58.00%	58.00%	0.552
2019+2018	Random Forest	57.49%	57.50%	43.80%	57.30%	57.50%	57.30%	0.542
2019+2018	AdaBoostM1	53.62%	53.60%	48.80%	52.90%	53.60%	52.90%	0.554
2019+2018	SMO	55.56%	55.60%	45.40%	55.50%	55.60%	55.50%	0.551
2019+2018	Multilayer Perceptron	53.62%	53.60%	47.50%	53.50%	53.60%	53.50%	0.548
2019+2018	Naïve Bayes	51.69%	51.70%	47.30%	52.60%	51.70%	51.70%	0.548
2020	J48	64.23%	64.20%	64.20%	-	64.20%	-	0.467
2020	Random Forest	64.23%	64.20%	64.20%	-	64.20%	-	0.486
2020	AdaBoostM1	64.23%	64.20%	64.20%	-	64.20%	-	0.468
2020	SMO	64.23%	64.20%	64.20%	-	64.20%	-	0.500
2020	Multilayer Perceptron	64.23%	64.20%	64.20%	-	64.20%	-	0.536
2020	Naïve Bayes	51.22%	51.20%	38.20%	61.40%	51.20%	50.90%	0.482
2020+2019	J48	59.38%	59.40%	56.70%	57.00%	59.40%	48.40%	0.529
2020+2019	Random Forest	59.69%	59.70%	56.10%	58.00%	59.70%	49.40%	0.548
2020+2019	AdaBoostM1	58.75%	58.80%	59.30%	34.80%	58.80%	43.70%	0.519
2020+2019	SMO	59.06%	59.10%	56.50%	55.70%	59.10%	49.10%	0.513
2020+2019	Multilayer Perceptron	57.50%	57.50%	56.40%	52.80%	57.50%	49.80%	0.545
2020+2019	Naïve Bayes	52.81%	52.80%	40.70%	58.70%	52.80%	51.90%	0.532
2020+2019+2018	J48	57.58%	57.60%	57.80%	50.50%	57.60%	44.50%	0.518
2020+2019+2018	Random Forest	58.48%	58.50%	55.90%	56.20%	58.50%	47.70%	0.539
2020+2019+2018	AdaBoostM1	57.88%	57.90%	58.40%	33.80%	57.90%	42.70%	0.498
2020+2019+2018	SMO	58.79%	58.80%	54.70%	56.90%	58.80%	49.70%	0.520
2020+2019+2018	Multilayer Perceptron	54.85%	54.80%	58.50%	46.80%	54.80%	45.50%	0.533
2020+2019+2018	Naïve Bayes	51.82%	51.80%	43.20%	56.10%	51.80%	51.10%	0.524

As reported in the answers to RQ1 and RQ2, we obtained values for some ML models close to 90%, which can be considered very good. However, the fact that the most recent datasets are the ones that usually present the best results, mainly the one from 2020, is a good indicator of the feasibility of future improvement, by further advancements in the methodological process, which has been progressively refined each year. Thus, the answer to this RQ is yes, it is possible to use collective knowledge for CS detection.

3.4.3.1 Summary of RQ3 results

The answer to this RQ is yes, it is possible to use collective knowledge for CS detection. The *Crowdsmelling* approach obtained values for some ML models close to 90%, which can be considered very good. Overall our results are similar to those of Di Nucci et al. [31]'s study and are even better in some metrics. The fact that the most recent datasets are the ones that usually show the best results, mainly the one from 2020, is a good indicator of the feasibility of future improvement.

3.5 Discussion

3.5.1 Research Questions (RQ)

In this section, we present the discussion of the results considering the three RQs. Regarding the comparison of our results with existing works, we will compare with [31] study, since it is the one that presents more similarities with ours, also using more realistic datasets.

For the *Long Method*, the model with the best prediction is *AdaBoostM1*, trained on the 2020 dataset, with a *ROC* of 0.870 (see Table 3.3), but also *F-Measure* and *Accuracy* show values higher than 80%, namely 81.50%, and 81.36%, respectively (see Table 3.6). *Random Forest*, with a *ROC* of 0.869, shows a value almost equal to *AdaBoostM1*. These two models show good results, in line with the results presented in Di Nucci et al. [31]'s study. The *Multilayer Perceptron* and *J48* models also show good results with the best *ROC* of 0.868 and 0.832, respectively, for the 2020 dataset. Namely, *Multilayer Perceptron* is the second-best model for three datasets (2019+2018, 2020+2019, 2020+2019+2018). In opposition is *Naïve Bayes* and *SMO*, which show the worst results for all datasets, for example, for the 2020 dataset where they have their best values, the *ROC* is 0.783 for *Naïve Bayes*, and 0.803 for *SMO*.

Also, regarding the *Long Method*, the models trained with the most recent dataset, the year 2020, have the best values, with a *ROC* greater than or equal to 0.803 for five models (*J48*, *Random Forest*, *AdaBoostM1*, *SMO*, *Multilayer Perceptron*) out of the six we used. Only the *Naïve Bayes* model has a *ROC* lower than 0.800, more precisely 0.783, but still higher than all models trained with datasets from previous years. The fact that the models trained on the most recent dataset show the best results is important because it means that there has been an evolution in the production of the datasets over the three years by this approach.

For the *God Class*, prediction values very close to 90% were obtained. As such, we consider these to be good values compared to similar studies. The model that presented the best ROC value was *Naïve Bayes* with 0.896, followed by *Random Forest* with 0.893 (see Table 3.4) for the 2020 dataset. These two models also had the best values for the other metrics, with both having equal values for *F-Measure* 88.70% and *Accuracy* of 88.97% (see Table 3.7) for the 2020 dataset. For the *Multilayer Perceptron* model, good results were also obtained, with a *ROC* of 0.885, for the 2020 dataset and a *ROC* of 0.880 for the 2020+2019+2018 dataset. This model presented the third and fourth best values. The *AdaBoostM1* and the *SMO* models obtained their best values with the 2020 dataset, with a *ROC* value of 0.876 and 0.857, respectively. The worst values were presented by the *J48* model, with its best *ROC* value of 0.791, thus being the only model that failed to exceed the *ROC* value of 0.800. For the code smell *God Class* it happened the same as for the code smell *Long Method*, all models presented their best *ROC* values when trained with the most recent datasets (from 2020).

For the *Feature Envy*, it was not possible to obtain the values for all the evaluation metrics for the aforementioned reasons (see subsection 3.4.2.3). The models for this code smell showed low results, being the worst results of the three CS. Thus, the best ROC value was 0.570 for *Random Forest*, but far from the values obtained for *God Class* and *Long Method*, 0.896 and 0.870, respectively. The *Naïve Bayes* model showed the best result of all the evaluation metrics with a value of 61.40% for *Precision*. *Random Forest* again presented the best value for *Recall* 59.70%

and *Accuracy* 59.69%. For *F-Measure*, the best value of 58.00% is obtained with the *J48* model. Regarding the datasets that show better results for *Feature Envy*, the *Accuracy*, *Precision*, and *Recall* metrics were the 2020+2019, for the *F-Measure* and *ROC* metrics were the 2019+2018 and 2019, respectively. Hence, no dataset concentrates most of the best values for the various metrics.

3.5.1.1 RQ1. What is the performance of ML techniques when trained with data from the crowd?

The best result was obtained for the *God Class* with a *ROC* value of 0.896, however, the *Long Method* with a *ROC* of 0.870 is very close. The worst result was obtained for *Feature Envy* with a *ROC* of 0.570. The difference in *ROC* value between the best and worst code smell is 0.326. This considerable difference is due to the composition of the *Feature Envy* datasets. When we analyze it, we see that the diversity of cases (classified methods) is much smaller compared to that of the *Long Method* datasets, which is also a code smell in the method scope, and consequently uses the same code metrics. The solution to this problem is to continue to grow the dataset by classifying methods that are not already part of it. However, this problem alerts us to the classification of more complex CS, as such, with fewer occurrences in the code and where programmers tend to follow more of the advisors' detection results.

When we compare our *ROC* values with those obtained by Di Nucci et al.[31], we find that for the CS *God Class* and *Long Method*, we obtain similar values in the range of 0.89 and 0.87, respectively. Regarding the code smell *Feature Envy*, for the reasons already presented, our value of 0.57 is considerably lower than the one presented by Di Nucci et al.[31], which is 0.89.

3.5.1.2 RQ2. What is the best ML model to detect each one of the three CS?

Having the *ROC* as the reference metric, for the *Long Method*, the best models were *AdaBoostM1* and *Random Forest*, for the *God Class*, it was Naive Bayes and *Random Forest* that presented the best values, and for the *Feature Envy*, it was *Random Forest* and Naive Bayes models. Thus, we can conclude that regarding which is the best ML model for the detection of the three CS, we do not have a model that guarantees the best detection value in the three smells, however, *Random Forest* stands out.

When we compare the results of the models for the detection of the three smells, we verify that the best results are obtained by the *God Class* detection models, and the *Feature Envy* detection models obtain the worst results.

In the Di Nucci et al.[31] study, the best performances (for all CS) were obtained by the *Random Forest* and *J48* models. These two models have in common that they are based on decision trees. When we compare them to our models, we can conclude, i) *Random Forest* was also the model with which we obtained the best results when considering all smells, ii) regarding *J48*, it was not one of our best models, because only for the *F-Measure* in the *Feature Envy* code smell it presented the best value, iii) Naive Bayes, which was one of our best models, did not present significant results in the Di Nucci et al.[31] study.

3.5.1.3 RQ3. Is it possible to use Collective Knowledge for CS detection?

Crowdsmelling is a pioneering approach for code smell detection. As such, there are always methodological aspects that can be improved, such as the ones we present in this discussion.

When we have dozens of participants, it is not possible to have total control over the actions of each participant. In our case, this was reflected in the non-use, in the first year, of data from 11 teams, out of a total of 42 participants. To have better control over the participants' actions, we removed the possibility for them to choose the Java project on which to detect the CS, and all the teams started using the same project. This decision resulted in a lack of diversity of cases when a code smell is more complex and consequently has fewer existences in the code (in our case, *Feature Envy*). Another consequence was that the participants started to follow the advisor's suggestions more since this code smell is more complex then *Long Method* and *God Class*.

In this experiment, we performed many processes manually, such as the data aggregation process, which were very time consuming tasks and, therefore, impractical to implement in a company's reality. However, our goal was to perform the first experiment to verify the potential of this approach.

This first study presents promising results, therefore, corroborating the feasibility of *Crowdsmelling*. It allowed us to gather in the datasets a wide variety of opinions (than 350 participants) regarding the classification of CS. Another reason is that the datasets have borderline smells (where it is not clear whether it is a smell or not), making them harder to detect. Finally, the fact that the datasets are not balanced also contributes to a more realistic setup.

We have organized the datasets by year to compare the results of each year and thus understand the progress in implementing this approach. The results obtained show progress over these years, with the best values being obtained in 2020. Thus, we are led to conclude that we are on the right track and that we can improve these results much more.

From a methodological point of view, more validation experiments are needed to cover more CS, build more broad datasets, and increase external validity. To address many of the issues presented, a microservice-based architecture to automate the whole process of the *Crowdsmelling* approach, from the extraction of metrics from the Java project code to the validation of CS by the developers is proposed in chapter 5.

3.5.2 Implications and limitations of the *Crowdsmelling* Approach

The *Crowdsmelling* approach has several advantages for developers and researchers because it is a dynamic approach that does not require the definition of rules for the detection of each code smell and its thresholds. Through the input given by developers, this approach produces datasets more and more adapted to the developers' reality, which implies the production of better ML models and, consequently, better detection of CS. This dynamics presented by the approach has two main advantages: i) although we have used only three CS in this study, it is not limited to these CS and can be generalized to other CS; ii) it makes the detection *Accuracy* improve as the feedback from the developers grows (by improving the learning datasets) and leads the ML models to converge to maximum *Accuracy*.

The learning dynamics presented in the previous paragraph is also the main limitation of the approach because it depends on developers' feedback, and it is not possible to predict exactly how much convergence in learning can be achieved, i.e., what is the maximum detection *Accuracy*.

To better demonstrate our approach, we will exemplify two scenarios:

(i) A company where there is a set of development rules, namely, CS, that is known and respected by all developers. In this scenario, all developers are aligned with the CS rules, thus contributing to a clear definition of what a code smell is in the datasets. In this scenario, we have faster convergence and will achieve higher detection *Accuracy*;

(ii) A successful open-source project, where many developers contribute. In this scenario, it will be more complicated for all developers to respect the rules since there will be less alignment among developers, and therefore more divergence on recognizing the occurrence of each code smell type. Our approach will always translate in the datasets what the developers understand to be a code smell, but the convergence will take longer, and the detection *Accuracy* will be lower.

In both scenarios, the *Crowdsmelling* approach learns from the context in which it is used by learning the CS detection rules used, thus always translating the developers' reality. The more precise the detection rules, the higher the detection *Accuracy*.

3.5.3 Threats to validity

In our study, we made assumptions that may threaten the validity of our results. This section discusses possible sources of threats and how we mitigated them.

3.5.3.1 Conclusion Validity

Threats in this category impact the relation between treatment and outcome.

The first threat is in the evaluation methodology, so we adopted the 10-Fold Cross-Validation, which is one of the most used in ML, and directly compared our results with those achieved in the other studies.

As for the evaluation metrics adopted to interpret the performance of the experimented models, we have adopted the most common ML metrics, which have been used in other studies with some similarities.

To test if there was a statistically significant difference between the performance of the six ML models, we used the one-way analysis of variance (ANOVA).

3.5.3.2 Construct Validity

As for potential issues related to the relationship between theory and observation, we may have been the subject of problems in the adopted methodology. To avoid bias in the process, we elaborated a script in which we detailed all the steps that the teams had to carry out to detect CS so that there would be uniformity in the process. However, we cannot guarantee the correct use of this script by all the teams.

Code metrics are vital because they play the role of independent variables in ML algorithms. To avoid bias in metrics extraction, we used the same metrics as in Fontana et al. [6], since they are publicly available.

As for the experimented prediction models, we exploited the implementation provided by the Weka framework [51], which is widely considered a reliable tool. To avoid bias in the parameterization of the Weka algorithms, we used the default values for the parameters.

3.5.3.3 Internal Validity

This threat is related to the correctness of the experiments' outcome. Since the definition of CS is subjective, it may cause different interpretations, so the manual evaluation is not entirely reliable. To mitigate this problem, an advisor is used in the experiment to serve as a basis for identifying CS, although each team was always in charge of the final decision.Teams were composed of several developers, and all had the same training.

To avoid participants only classifying CS detected by the *JDeodorant* advisor (although it was optional to use it), it was indicated that they would have to classify at least one package, however, not all teams did so. This required teams to manually classify, based directly on code metrics, a set of false positive and negative CS detected by *JDeodorant*.

The participants in this study were students attending a compulsory Software Engineering course. In the scope of this course, an optional assignment was done where this experiment was carried out. To have rigor in the accomplishment of this work, since it was optional, the works were evaluated by the teachers, and a grade was assigned according to the quality demonstrated. The fact that only students were used can be a threat, however, these are finalists who in three months will be working in companies. On the other hand, the use of students has advantages and disadvantages, as we can see in the paper by Feldt et al. [36].

The team members' maturity, experience, and knowledge about CS is a variable that we cannot control. As such, there may be variations in the *Accuracy* and *Precision* of CS detection. To minimize the possible bias, the decisions were not individual but taken by the team. The time given to do this work was three weeks, which may have been a reason for bias, but we thought it was sufficient considering it was a team effort.

Because CS are only detected in three Java projects, there may be some bias about the number and type of CS existing in these Java projects. However, we chose these projects because they are open-source, are widely used in CS detection, and are not toy examples due to their considerable dimension.

3.5.3.4 External Validity

Finally, external validity is concerned with whether we can generalize the results outside the scope of our study.

With respect to generalizability, we used the three most common CS in this type of study. Regarding the code metrics, we used a high number, 61 metrics for *God Class* and 82 metrics for *Feature Envy* and *Long Method*, thus ensuring a broad scope.

In terms of programming languages, we only used Java projects, but Java is by far the most used language in CS detection studies, accounting for 77.1% of the cases (see section 2.4.4).

The fact that this study has a manual component hampers its replication. However, all the necessary indications are in the study, and a set of materials is available on GitHub[9] and Zenodo [112].

3.6 Summary

Main conclusions. We have proposed the *Crowdsmelling* approach – use of collective intelligence in the detection of CS – to mitigate the problems described above of subjectivity and lack of calibration data required to obtain accurate detection model parameters. *Crowdsmelling* is a collaborative crowdsourcing approach based in ML, where the wisdom of the crowd (of software developers) will be used to collectively calibrate CS detection algorithms. In this chapter, we reported the results of a study investigating the feasibly of its application.

For three years, we collected CS detection data by several teams manually, although they could use *JDeodorant* as an advisor if they wanted. Combining the data from each year with the previous ones, we created several oracles for each of the three CS (*Long Method*, *God Class*, *Feature Envy*). The latter was used to train a set of ML algorithms, creating the detection models for each of the three CS, in a total of 108 models. Finally, to evaluate the models, we tested them using the 10-Fold Cross-Validation methodology and analyzed the metrics *Accuracy*, *Precision*, *Recall*, and *F-Measure*, with particular emphasis on *ROC*, because the datasets were not treated, for example, balanced. This way, we created the most realistic datasets possible. To check if there were significant differences between the classifications presented by the different models, we proceeded to the analysis of variance through a one-way analysis of variance (ANOVA).

Regarding RQ1, we conclude that the *Random Forest* and *AdaBoostM1* algorithms obtained the best results for the code smell *Long Method*. The best result with a *ROC* of 0.870 was obtained by *AdaBoostM1* when trained by the dataset 2020, followed by the *Random Forest* with a *ROC* of 0.869 for the same dataset. The best result for the code smell *God Class* was obtained by the *Naïve Bayes* algorithm, when trained by the dataset 2020, with the *ROC* value of 0.896. For *Feature Envy*, the results are low, with the *Random Forest* algorithm having the best *ROC* value of 0.570 when trained by dataset 2019. The results of the variance tests (ANOVA) show there was no statistically significant difference between the performance of the six ML models when trained with data from the crowd and, therefore, more realistic.

As for RQ2, the best ML model for *Long Method* detection is *AdaBoostM1*, presenting the best values for all evaluation metrics, a *ROC* value of 0.870, an *Accuracy* of 81.36%, a *Precision* of 82.90%, a *Recall* of 81.40%, and *F-Measure* of 81.50%. For the *God Class*, the model that presents the best value for the *ROC* is Naive Bayes, with a value of 0.896. the Naive Bayes and *Random Forest* models present an *Accuracy* value of 88.97%, a *Precision* value of 89.70%, a *Recall* value of 89.00%, and an *F-Measure* value of 88.70%. For the *Feature Envy*, the best model is the *Random Forest* with a *ROC* of 0.570. However, the best performance in the detection of *Feature Envy* is obtained by the Naive Bayes model for *Precision* with a value of 61.40%.

Regarding RQ3, it is possible to use *Crowdsmelling* – use of collective intelligence in the detection of CS – as a good approach for the detection of CS because we obtained values for some

[9] https://github.com/dataset-cs-surveys/Crowdsmelling

ML models close to 90%, which can be considered very good, for realistic datasets, which reflect the detection performed by developers. The fact that the most recent datasets, the year 2020, are the ones that usually present the best results leaves us with great motivation to continue developing this detection approach because we think that we can even better the results.

The results suggest that *Crowdsmelling* is a feasible approach for the detection of CS.

Chapter 4.

Code Smells Visualization

Contents

4.1 Introduction **82**
4.2 Visualization Survey **82**
4.2.1 Survey and Samples 83
4.2.2 Survey Results 83
4.3 *Smelly Maps* as SourceMiner Views **89**
4.4 Summary **91**

In this chapter, we present our approach to visualizing code smells, *Smelly Maps*, and the survey results of the community working on software visualization.

4.1 Introduction

Code smells awareness features should be in place in the IDE itself since that is the programmer's workbench. These features are particularly important to spot code smells in large, complex software systems, where their distribution and collateral effects may spread a lot. Effective, yet non-intrusive, visualization features should allow to (i) spot the location of code smells, (ii) diagnose their cause, and (iii) warn of their potential hazardous consequences. The latter requirement seems trivial since there is abundant corroborating published evidence that can be used for providing contextual warnings. As for the former two are much more demanding since they require finding out which are the adequate levels of granularity to visualize code smells to their full extent.

Code smells visualization features should be built on top of software visualization grounds since we need to superimpose information regarding the smell on top of the software structure visualization itself. We may see it as something that resembles augmented reality, where the "reality" here is the code itself being developed or maintained.

Many software visualization metaphors have been proposed in the literature [32]. Consider, for instance, one of the most sophisticated ones: a software system as a city [139]. There, packages are represented as neighborhoods, classes as buildings, and methods as floors of those buildings. In such a setup, it would not be upfront to represent, for instance, code smells that occur within a hierarchy, simply because class hierarchies are not clearly mapped into this appealing city metaphor. This counterexample highlights the need to further research to determine which metaphor is the most appropriate for each code smell.

In our SLR on chapter 2, we classified CS visualization techniques into two categories (see 2.4.12: (i) the detection is done through a non-visual approach, the visualization being performed to show CS location in the code itself, (ii) the detection is performed through a visual approach. In this chapter, we present our proposal of feature work - *Smelly Maps* - to visualize code smells, which falls into the first category. Our proposal first detects the code smells with the *Crowdsmelling* approach and then visualizes them using the Smelly Map corresponding to the type of code smell.

Finally, we present the results of a survey on visualizing code smells conducted among the academic community.

4.2 Visualization Survey

As mentioned in the validation of the SLR (see section 2.5.2), we conducted three surveys. A pre-test, a survey addressed to the authors of the studies that are part of the SLR presented in chapter 2, and the third one was addressed to the software visualization community. These surveys aimed at understanding the perception and opinion of these communities about the detection and visualization of code smells. In the SLR validation section, we presented a summary of the results of the second and third aggregate surveys. In this section, we look in more detail at the results from the part of the survey on visualization of code smells.

4.2.1 Survey and Samples

The questionnaire consists of a Cover Letter and a Consent Information Letter, followed by two parts, Part I - Code smells visualization and Part II - Code smells detection. This section will only look at the part about code smells visualization, which corresponds to 6 questions.

The answer to each question consists of 3 components:

1. The answer itself, on a 6-point Likert scale (Strong disagreement, Disagreement, Weak disagreement, Weak agreement, Agreement, Strong agreement);

2. A slider between 0 and 4 that measures the degree of confidence of the answer;

3. An optional field to describe the justification of the answer or for comments.

The subjects invited for the survey were researchers with the most relevant work in code smells detection and software visualization. Regarding code smells detection, we chose to send the questionnaire to the 193 authors of the papers that were part of the SLR presented in chapter 2.Regarding software visualization community we invited the 380 authors of the papers from the SLR by Merino et al. [85] on software visualization that consists exclusively of papers presented at the SOFTVIS and VISSOFT conferences. Besides sending the link to the survey to the aforementioned 380, we also publicized the survey through a post on the software visualization blog[1]. A total of 74 subjects answered these questionnaires. However, most did not answer all the questions. The most answered question had 39 responses, and the least answered question had 23 responses.

The survey was conducted online in the first half of 2019.

The structure of the surveys, collected responses, and descriptive statistics on the latter are available at a GitHub repository[2] and Zenodo [111].

4.2.2 Survey Results

The survey responses confirm the conclusions obtained through the SLR, presented 2.6 section. As far as visualization is concerned, we will look at each of the questions in more detail below. These questions correspond to SLR finding F11, and the results are presented in the Table 2.15.

In each of the questions, we will include some of the comments inserted by the researchers because they complement the answers and reveal what they think of the question's theme.

4.2.2.1 Question - The vast majority of code smells detection studies do not propose visualization features for their detection

This question showed the highest level of agreement with the conclusions we drew from the SLR. To this question, 66.7% of the participants showed agreement and 15.4% strong agreement (see Figure 4.1). In addition, the confidence level with which they responded was also high, 3.0, with a standard deviation of 1.0. It is also worth noting that this question received the highest number of responses from the academic communities.

[1] https://softvis.wordpress.com/
[2] https://github.com/dataset-cs-surveys/Dataset-CS-surveys.git

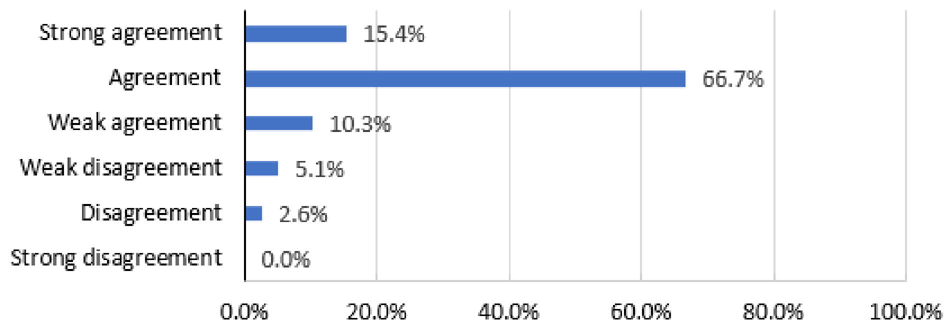


Figure 4.1: Answers to the question: "The vast majority of code smells detection studies do not propose visualization features for their detection"

When we add up the answers corresponding to "Strong agreement"and "Agreement,"we get 82.1%, which provides a sound evidence that the absolute majority of the studies only do code smells detection and do not present visualization strategies.

Comments inserted by the researchers to this question:

1. *Detection and visualization are separate issues. A paper should generally focus on only one of these aspects (separation of concerns holds true also for research papers). Papers are limited in space and generally you cannot do a good job in covering both aspects. For clone detection, there are quite a few visualization papers. Moreover, first things first. First we need to solve the problem of detection code smells, then only we can try to visualize them. Hence, it may not be surprising that we have fewer visualizations than detection techniques at this stage of research maturity.*

2. *It depends by what we mean by "visualization". I am not sure that developers would need such features (if they would, the features would be implemented).*

3. *Depends what you mean by visualization. Is a simple scatterplot or bar chart also a visualization? If not (i.e. we consider only more 'advanced' visualizations), then I am quite confident about my answer.*

4.2.2.2 Question - The vast majority of existing code smells visualization studies did not present evidence of its usage upon large software systems

Figure 4.2 shows the percentages of responses to this question. We can see that the option with the highest number of answers is "Agreement"with 43.8%, followed by "Weak agreement"with 34.4% and "Strong agreement"with 12.5%. The confidence level of the answers is 2.9, which is about average, which is 3.

The visualization of code smells in large systems is an important point due to the complexity of the systems and some code smells, especially those that cross several classes. However, it should be noted that in the SLR, we only found three studies [S7, S17, S16] with solutions dedicated to large systems.

Comments inserted by the researchers to this question:

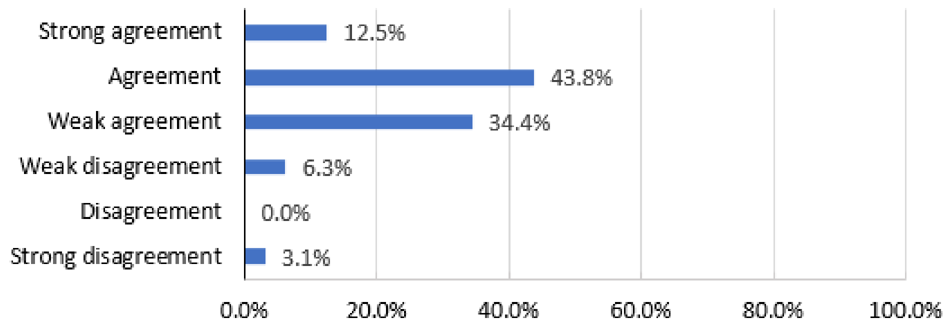


Figure 4.2: Answers to the question: "The vast majority of existing code smells visualization studies did not present evidence of its usage upon large software systems"

1. *it would be better to have an option called I do not know because for this question for instance I don't have the right answer.*
2. *To be honest, I don't know.*
3. *Do not know enough code smells visualization studies to make a statement on that.*
4. *Unfortunately, that is true for most visualization papers. It is certainly true for visualization of software clones. We are currently working on a systematic mapping study on clone visualization and have found hardly any kind of empirical evaluation of the proposed visualization techniques.*
5. *It is difficult to properly assess a visualization approach.*
6. *I cannot answer the question because I only know few of the respective visualizations.*
7. *Depends what you mean by 'large': 10K lines of code? 100K? More than 1 million? If more than say 200..400K, then I would agree..strongly agree with the question.*

4.2.2.3 Question - Software visualization researchers have not adopted specific visualization related taxonomies, such as the ones below, to support the identification of code smells: B. Price, R. Baecker, I. Small, A principled taxonomy of software visualization, Journal of Visual Languages and Computing 4 (3) (1993) 211–266. Roman, G. C., & Cox, K. C. (1993).A taxonomy of program visualization systems. Computer, 26(12), 11-24. Maletic, J. I., Marcus, A., & Collard, M. L. (2002). A task oriented view of software visualization. In Proceedings First International Workshop on Visualizing Software for Understanding and Analysis (pp.32-40). IEEE. Gallagher, K., Hatch, A., & Munro, M. (2008). Software architecture visualization: An evaluation framework and its application. IEEE Transactions on Software Engineering, 34(2), 260-270. Myller, N., Bednarik, R., Sutinen, E., & Ben-Ari, M. (2009). Extending the engagement taxonomy: Software visualization and collaborative learning. ACM Transactions on Computing Education (TOCE), 9(1), 7.

This question obtained the lowest level of agreement, with the most voted answer being "Weak agreement"with 46.9%, followed by "Agreement"and "Strong Agreement", respectively with

28.1% and 9.4% (see Figure 4.3. However, it should be noted that this was the answer with the lowest confidence level, 2.0.

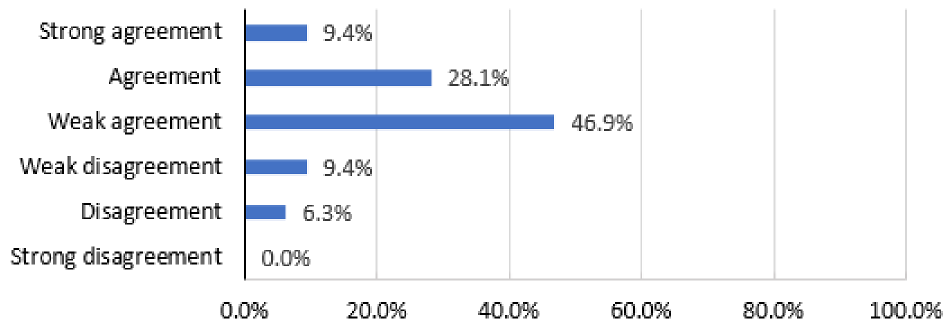


Figure 4.3: Answers to the question: "Software visualization researchers have not adopted specific visualization related taxonomies to support the identification of code smells"

Looking at figure 4.3, we can conclude that the adoption of taxonomies by software visualization researchers is not yet a common practice.

Comments inserted by the researchers to this question:

1. *Don't know.*

2. *Do not know enough about Software visualization studies wrt. code smells to answer this question.*

3. *Why should one adopt a visualization taxonomy for *detection* of code smells? Visualization comes after detection. I am neither sure what it means precisely to adopt a visualization taxonomy. Do you mean the researchers should classify their new visualization into one of those taxonomies?*

4. *Not convinced that the referenced papers are (still) relevant. I know the one of Maletic et al. - it's okay, but does not go into details. I don't use it for my own papers.*

5. *I know of quite a large number of software vis papers (and researchers) who have adopted at least one of the above mentioned taxonomies.*

4.2.2.4 Question - If visualization related taxonomies were used in the implementation of code smells detection tools, that could enhance their effectiveness.

Most participants split between the response "Agreement"and "Weak agreement", both with 38.2%. The third most chosen response was "Strong agreement"with 11.8% (see Figure 4.4). The confidence rating of the responses was 2.8, very close to the overall average of 3.

We can conclude that while the majority agree that if visualization-related taxonomies were used in the implementation of code smell detection tools, this could increase their effectiveness, there is no firm agreement.

Comments inserted by the researchers to this question:

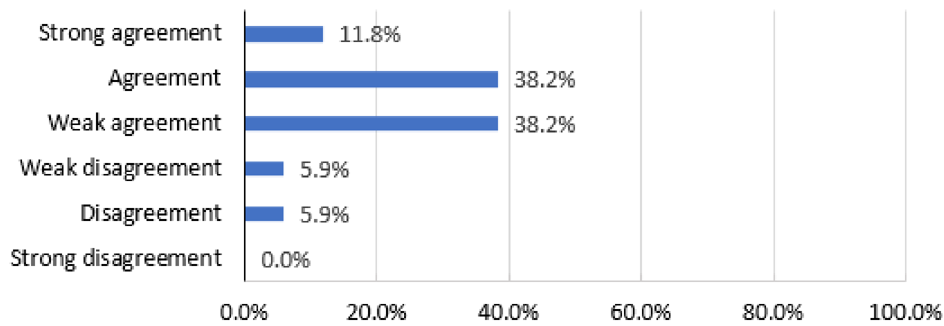


Figure 4.4: Answers to the question: "If visualization related taxonomies were used in the implementation of code smells detection tools, that could enhance their effectiveness."

1. *Simply, each problem has its own characteristics and the problem of smell detection would deserve independent studies aimed at understanding how developers would actually visualize them.*

2. *There is a need to show the symptoms of goat smells and visualization can definitely help.*

3. *Definitely. The taxonomies, to prove their usefulness, have to be implemented in tools.*

4. *Do not know enough about Software visualization wrt. code smells to answer this question.*

5. *Maybe it could help to think about the requirements and properties of those new visualizations. Quite likely, a groundbreaking new visualization is not even necessary. There are already so many ideas that might be used. My impression is that there is generally very little space left for truly new types of visualization. There are only small incremental improvements and generally only adaptation and combinations of existing visualization techniques. The question must be asked what radically new requirements need to be fulfilled for visualizing code smells going way beyond the requirements for visualization of software aspects in general.*

6. *Probably, yes. I think the main problem is that software engineering researchers lack of HCI experience.*

7. *Taxonomies, in general, can help to compare approaches, not sure they are that helpful for developing tools/approaches.*

8. *I'm not sure that a taxonomy is directly contributing to an implementation as such. A taxonomy helps to organize related work, ideas, etc, and present/frame one's own work, but doesn't help directly when implementing a new method.*

4.2.2.5 Question - Which of the following visual attributes have you implemented in tools targeting the support of code smells identification?

Regarding the visual attributes implemented, this question received the least number of responses, 23. This is mainly due to 2 factors: the question's specificity, and the fact that there are few visual approaches. As such, not many researchers have used visual attributes. Figure 4.5

presents the main resources discussed in the literature [83], and the percentage of implementation of the same in tools. We can see that the most used visual feature is Color: Hue used by 62.5% of the researchers, followed by Form: Collinearity used by 52.2% and by Spatial position: 2D and Color: Intensity, both with 43.5%.

Overall the most used visual resources are color and shape. In the case of shape, its properties include spatial grouping, size, length, shape, orientation, and width.

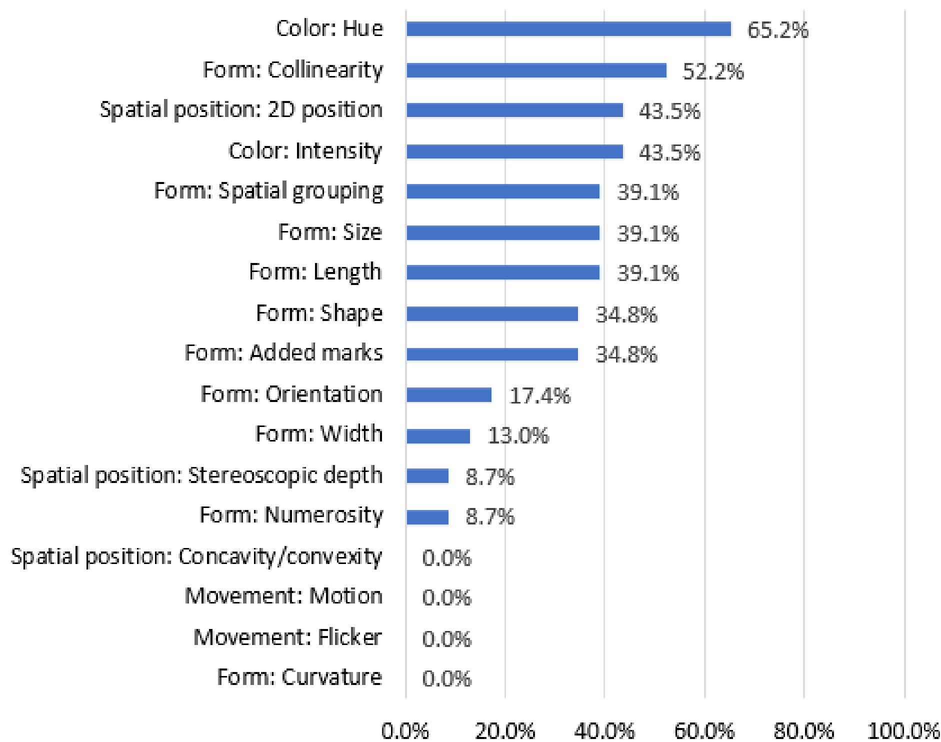


Figure 4.5: Answers to the question: "Which of the following visual attributes have you implemented in tools targeting the support of code smells identification?"

Comments inserted by the researchers to this question:

1. *N/A. I didn't implement visualization features in my smell detectors.*

2. *Not applicable: I have not implemented tools with visual attributes.*

3. *Do not know.*

4. *I have co-authored one publication in this area using Parallel coordinates and scatterplots. Of course, these visualization relate to the above encodings, but it's not clear in all cases how. The above encodings seem a bit too low-level to make an appropriate classification here.*

4.2.2.6 Question - The combined use of collaboration (among software developers) and visual resources may increase the effectiveness of code smells detection.

This question was the only one in which no disagreement was recorded, as can be seen in Figure 4.6. "Agreement"was the most given answer with a percentage of 50.0%. "Weak agreement"was answered by 26.5% of the subjects, and "Strong agreement"by 23.5%.

We can see the importance of collaboration between software developers and visual resources in answering this question. Of course, the detection of code smells is the principal component, but visualization is also important for a better understanding of the more complex smells.

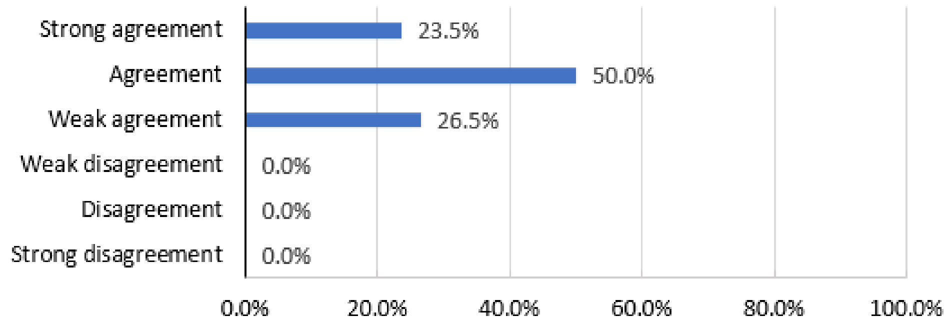


Figure 4.6: Answers to the question: "The combined use of collaboration (among software developers) and visual resources may increase the effectiveness of code smells detection."

Comments inserted by the researchers to this question:

1. *I strongly agree and if you need any further assistance or help I would be glad to support such research and efforts.*

2. *Well, both factors help, but I do not see an explicit synergistic effect between them.*

3. *It is a strong belief of mine, yet only a belief until we have empirical evidence.*

4. *People with visual impairment will have problems.*

5. *Not sure I understand the question.*

4.3 *Smelly Maps* as SourceMiner Views

Code smells are defined at different scopes, as represented in Table 4.1. A visualization feature for a code smell of the first kind (within one method) seems straightforward: the best way is adding some type of flag, usually in the code editor's margin, in the place where it was found. Even though classes may be large (i.e., spread across several screen heights), it is still acceptable to use the flagging technique for representing the location of code smells of the second kind (within one class), for instance, close to the class header. As for the other three kinds of code smells, we need to identify adequate visualization mechanisms since they may spread across a large number of methods or classes.

We hypothesize that at least as many visualization metaphors (or views) as the aforementioned code smells scopes are required. In the scope of this thesis, we will call *Smelly Maps* to these views. *Smelly Maps* will act as a front-end for the more complex code smells, facilitating the understanding of their side-effects and the diagnosis of their cure.

Based on a well-known reference model for information visualization from Card et al. [20], Carneiro and Mendonça extended and adapted it to the context of Multiple Views Interactive Environments (MVIE) [21], as portrayed in Figure 4.7.

Table 4.1: Different scopes of code smells

Scope	Code Smells from Fowler [43]
Within one method	Comments, Long Method, Long Parameter List, Primitive Obsession, Speculative Generality, Switch Statements, Temporary Field
Within one class	Comments, Data Class, Data Clumps, Large Class, Lazy Class, Speculative Generality
Within a class hierarchy	Inappropriate Intimacy, Refused Bequest
Across several methods	Feature Envy, Message Chains, Middle Man, Shotgun Surgery
Across several classes	Alternative Classes with Different Interfaces, Divergent Change, Duplicated Code

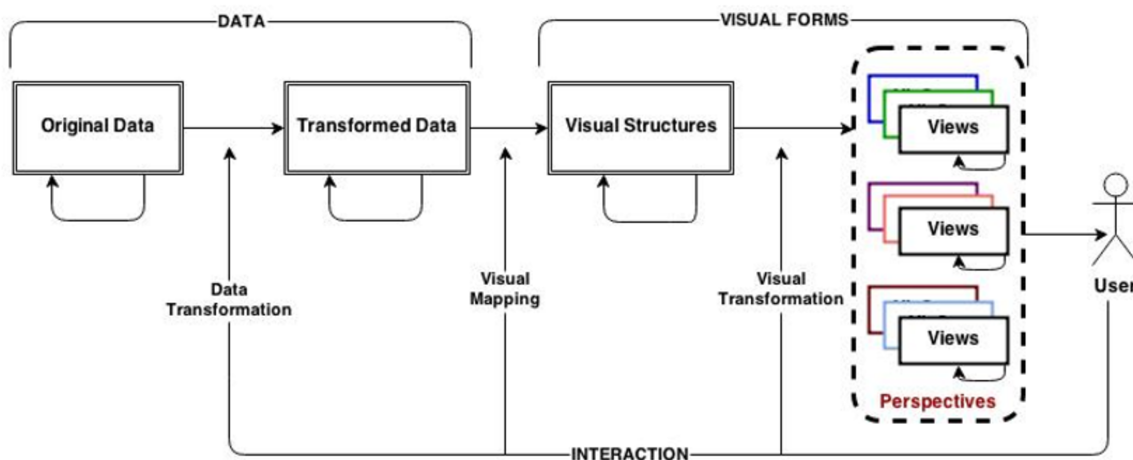


Figure 4.7: An extended reference model for MVIEs (from [23])

The process starts with original (raw) data obtained from a repository that undergoes a set of transformations, which is then organized into data structures suitable for information exploration. This process is called **data transformation**. Next, the aforementioned data structures are used to assemble visual data structures. Those structures organize data properties and visual information properties to facilitate the construction of visual metaphors. This step defines the mapping from real attributes – which are derived from the data properties (software attributes, in our case) – to visual attributes such as shapes, colors, and positions on the screen. This process is called **visual mapping**. The final step in Figure 4.7 is the visual transformation, aimed at drawing the information on the screen to produce the views.

The aforementioned extended reference model was used to build a Java MVIE called *SourceMiner*[3], which was implemented as an Eclipse plugin in the co-supervisor's team [21, 22]. We propose to offer *Smelly Maps* as a set of new views in SourceMiner, which will work in cooperation with the *CrowdSmelling Checke*r plugin described in chapter 5. SourceMiner allows visualizing software attributes at different levels of abstraction (packages, types, and operations). Our objectives for the *Smelly Maps* include the ability to: (i) represent the distribution

[3]http://sourceminer.sourceforge.net

of code smells evidence; (ii) show the coverage of the assessment process of code smells occurrences, as aforementioned; (iii) represent the intensity of each code smell occurrence (using a color scheme); (iv) interactively browse the whole system.

4.4 Summary

Code smells are defined in different scopes (Within one method, Within one class, Within a class hierarchy, Across several methods, Across several classes), which means that in order to visualize them, it is necessary to adapt the visualization features according to the scope of the code smell. Viewing a *within one method* and *within one class* code smells is simple, by just tagging, usually in the margin of the code editor where the CS was found. For code smells *Within a class hierarchy*, *Across several methods*, and *Across several classes* it is more complicated. For these last three cases, we need to identify adequate visualization mechanisms since they may spread across a large number of methods or classes.In large and more complex systems, the problem described takes on an even greater dimension.

To mitigate the problem described above, have the intention of developing the concept of *Smelly Maps*. *Smelly Maps* will act as a front end to the more complex code smells, making it easier to understand their side effects and diagnose their cure. *Smelly Maps* will be a set of new views in the *SourceMiner* (Java Multiple Views Interactive Environments (MVIE) [21]) toll, which will work in cooperation with the *CrowdSmelling* plugin.

In conjunction with the SLR, present in chapter 2, a set of surveys was conducted in order to validate the findings of the SLR. The surveys consisted of collecting the degree of agreement of the academic community with the set of findings of the SLR. In this chapter, we presented the results in more detail of the visualization part of the survey. The degree of agreement between our findings and the opinion of the survey participants is high and was sometimes highlighted in the written comments in the survey.

Some of the conclusions are: most studies of code smell detection do not present visualization; for large systems, the scenario is even worse, as we only found three studies that refer to large systems; the use of taxonomies and other visualization features, when implemented together with detection, may increase the effectiveness of the CS detection tools.

This survey validates our proposal for visualization using *Smelly Maps* as a complement to our *Crowdsmelling* detection approach.

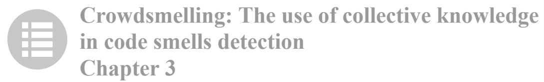
[This page has been intentionally left blank]

Part III.

Crowdsmelling: a ML-based crowdsourcing approach for code smells detection

PART I: FUNDAMENTALS

PART II: CODE SMELLS DETECTION AND VISUALIZATION



PART III: CROWDSMELLING: A ML-BASED CROWDSOURCING APPROACH FOR CODE SMELLS DETECTION

PART IV: CONCLUSION

This part implements the *CrowdSmelling* approach in a tool and scenarios of its use to detect code smells.

CHAPTER 5.

Crowdsmelling Tool

Contents

5.1 Introduction **96**
5.2 Motivation **96**
5.3 Related work **96**
5.3.1 Code smells detection tools 97
5.3.2 ML-based code smells detection 98
5.4 Crowdsmelling **100**
5.4.1 Proposed approach 100
5.4.2 Proposed architecture for an application using approach 101
5.4.3 Application usage scenarios 102
5.5 Summary **106**

This chapter proposesa tool for crowdsourcing approach based on supervised machine learning (ML) algorithms to mitigate the subjectivity problem of detecting code smells. We propose implementing the *CrowdSmelling* approach in a tool and presenting its use scenarios to detect code smells.

5.1 Introduction

The first results of applying *CrowdSmelling* suggest it is a viable approach (see chapter 3). In this chapter, we present a tool to support it. Developers use it through an Eclipse IDE plugin connected to a microservices architecture, composed of a scientific workflow management system (Taverna 2), an ML algorithms execution platform (Weka), and a database management system that communicate through REST interfaces.

The front-end of the proposed tool is a plugin installed in each developer's IDE that computes metrics from the source code and sends them to the back-end, requesting the location of detected code smells.

Crowdsourcing is fundamental in this approach since the feedback received from the crowd of software developers on false negatives and true and false positives is used to train multiple ML algorithms, allowing the dynamic calibration and choice of the best alternative for the detection of code smells.

This chapter will present the latest version of the code smells detection tool that implements the *CrowdSmelling* approach under development. Appendix C shows the evolution of the architectures of the *CrowdSmelling* tool versions.

5.2 Motivation

There are several tools for detecting code smells (see section 5.3), but often there is no consistency among then regarding their results [37, 109, 114] for the same code smell. One of the causes of getting different detection results is the detection technology used.

One of the most used approaches in smells detection tools is rule-based, so it is necessary to define the rules' thresholds, which is not an easy task. Logically, if we set different thresholds on rules that detect the same code smell, they will produce different detections. Therefore, many tools have let the user set their thresholds to mitigate this problem. However, this solution raises another problem, which is the knowledge that users need to have about code smells to set the thresholds.

ML-based approaches allow mitigating the problem of thresholds. However, the identification of code smells by ML-based approaches depends on the oracles used, namely for training and testing the algorithms. Unfortunately, there are few publicly available oracles, which represents a problem in ML-Based approaches.

This is where our motivation comes from, to build a tool that can reduce the subjectivity of code smells, produce oracles collectively, and perform the detection automatically through ML techniques. The reduction of subjectivity is achieved through a crowdsourcing process.

5.3 Related work

To the best of our knowledge, there are no crowdsourcing tools for automatic code smells detection, i.e., based on machine learning. Thus, this section describes the main tools available in code smells detection and techniques that use machine learning.

5.3.1 Code smells detection tools

Fernandes et al. [37] presented a systematic literature review (SLR) of CS detection tools. As a result of this SLR, 84 tools were discovered, 29 of which were available online for download. According to these authors, tools were available for nine programming languages, but the most significant number is for Java, C, and C++, with 56, 16, and 15 tools, respectively. Regarding the detection strategies used by the tools, 37% (31 out of the 84 tools) are metric-based, 13% (11 out of the 84 tools) use other detection strategies such as *Machine Learning* and *Logic Meta-programming*. It should be noted that the authors of this SLR were unable to discover the detection technique applied by 20% of the tools. Regarding the use of crowdsourcing in these tools, nothing is mentioned. Finally, a study comparing four tools (*inFusion*, *JDeodorant*, *PMD*, and *JSpIRIT*) in the detection of 2 code smells (*Large Class* and *Long Method*) was performed. The results of this comparative study indicated that the four tools showed redundant detection results due to the high agreement coefficient computed.

The only tool we know that uses crowdsourcing in code smells detection is presented by Paramita and Candra [102], but the detection is manual, with all the disadvantages that this technique presents, such as human-centric, tedious, time-consuming, and error-prone. Paramita and Candra [102] developed a code smell detection platform based on crowdsourcing, which they called CODECOD (an abbreviation of Code Smell Detection through crowdsourcing method), whose main objective is to facilitate and improve the quality of manual code smells detection. This platform decomposes a task (a source code file grouping in zip files) into microtasks (a single method or a single class), which are sent to the workers (someone who solves and understands the code). The quality control phase is based on a technique authors call *Find*, *Vote*, *Verify*, which is responsible for implementing the output agreement and majority vote methods to determine code smells candidates. The evaluation of the platform, performed by the authors, showed that using the CODECOD platform improves *Accuracy* compared to the traditional manual techniques.

We now describe other code smells detection tools without using a crowd-based approach.

Marinescu et al. [81] presented *iPlasma*, an integrated environment for quality analysis of object-oriented software systems. This tool detects code smell in Java and C++ and uses a metric-based detection approach. Like any approach of this type, it needs to calibrate thresholds in the detection rules. Thus, to establish the appropriate value of thresholds for detection strategy, they use DSTM (Detection Strategy Tuning Machine). This tool presents the advantage of calibrating thresholds specifically for a given development environment. The great disadvantage of this method is that it requires human feedback over a period of time.

JDeodorant [39, 132] is a tool that detects four code smells (*Long Method*, *God Class*, *Feature Envy*, and *Type-Checking*) and sugests refactoring opportunities.

Moha et al. [88] developed the DECOR tool that detects four antipatterns and 15 code smells, based on a symptom-based approach.

Stench Blossom is a smell detector proposed by Murphy-Hill [90] that uses an interactive ambient visualization to represent the smells detected in the code. The smells are represented by sectors in a semicircle, which the authors call *petals*. The direction, radius, and other attributes of the *petals* represent characteristics of the smells, such as type and strength.

Vidal et al. [134] proposed JSpIRIT (Java Smart Identification of Refactoring opportunITies), a tool that allows users to define their strategy to detect code smells, as well as prioritize their classification.

Palomba et al. [97] presented HIST (Historical Information for Smell deTection) which uses version control systems to look for changes in the code between different versions to detect five code smells. There are also other commercial tools, such as SonarQube [1], that work in the area of application maintenance, detecting code smells.

5.3.2 ML-based code smells detection

Machine learning-based approaches are more recent, Kreimer[69] being one of the first authors to use this type of detection technique. The most relevant studies that use ML algorithms to detect code smells are presented below.

Kaur et al. [60] presented a Review on Machine-Learning Based Code Smell Detection Techniques from the year 2005 to 2020. The main conclusions of the review were: more free ML techniques need to be made available so that researchers can evaluate their results and make comparisons more quickly; the code smells most used by researchers are Feature Envy, Long Method, God Class, and Blob; more open source applications are used in evaluating ML techniques than industrial projects, but it is essential to test the approaches with real-life datasets; few researchers use large datasets, being the most used ones small or medium-sized; Java is the most used programming language, and the use of ML approaches in other languages is still an open research area.

Caram et al. [19] conducted a Systematic Mapping Study on ML techniques for code smell detection and found that *Feature Envy* was the most commonly used code smell. Regarding ML algorithms, the most used is Genetic Algorithms (GA), used in 22.22% of papers, followed by *Naive Bayes Classifiers*. The reason GA is the most used is that it is used to optimize metrics, such as the calculation of thresholds. *Decision Tree* and *Random Forest* are the best performing algorithms.

According to Caram et al. [19], and Reis et al. [114] the most widely used ML algorithm in code smells detection were Genetic Algorithms (GA), mainly for defining smells detection rules and thresholds for rules. Mahouachi et al.[75] used GA to combine detection and correction steps to generate classification rules. Boussaa et al. [13] e Sahin et al.[117] presented GA-based solutions that propose the simultaneous evolution of 2 populations where the first one generates detection rules (based on code metrics) and the second one generates examples of code smells that are not detected by the first population. Thus, these two populations are evolved in parallel, producing the evolution of smells detection rules. Kessentini et al. [62] used a set of evolutionary algorithms, through *Parallel Evolutionary Algorithms* (P-EA), where each algorithm presents a different adaptation (fitness functions, solution representations, and change operators), with the common goal of performing smells detection. Mansoor et al.[78] used multi-objective genetic programming (MOGP) to identify code smells. The process consisted of using two sets of code examples, one with code smells and the other with well-designed code, to generate detection rules. The goal was to find the best combination of metrics that maximizes

[1] https://www.sonarqube.org/

the first set of examples' coverage and minimizes the second set's detection. Mkaouer [86] proposes the creation of detection rules adapted to the developer's preference. The technique uses a GP algorithm, which converges the detection rules to the developer's preference through feedback from the developer. The developer rejects or approves the code smells presented to him, using this feedback to evolve the algorithm. Saranya et al.[119] suggested for smell detection the use of similarity between the code being analyzed and a set of example defects. This similarity was calculated through the Euclidean distance-based Genetic Algorithm and Particle Swarm Optimization - EGAPSO.

Decision trees are one of the most used algorithms in ML approaches for detecting code smells, and it is noteworthy that the first algorithm used in ML approaches was a decision tree by Kreimer in 2005. Kreimer[69] used a decision tree (C4.5 algorithm) to detect the code smells Long Method and Big Class and developed an Eclipse plugin to perform this detection. Zibran and Roy [147] have developed a plugin for the Eclipse IDE, based on a suffix-tree, to detect type-1, type-2, and type-3 code clones. Rajakumari and Jebarajan [108] presented a technique based on the *Frequent Pattern Growth Method* (*FP-Growth*), where they construct an *FP-Tree*. The goal was the detection of type-1 and type-4 code clones. Amorim et al. [5] reported an experiment on smells detection via decision trees with the C5.0 algorithm. The features used in the construction of the tree were code metrics. The authors also concluded that if the features were selected using a Genetic Algorithm, the effectiveness of the Decision was improved. Kaur et al.[59] proposed a hybrid algorithm based on the *Sandpiper Optimization Algorithm (SPOA)* and the B-J48. *SPOA* will optimize the parameters of the B-J48 tree for the detection of 5 code smells.

Support Vector Machines (SVM) is another of the ML techniques used in code smells detection. Maiga et al.[76, 77] presented two studies that used SVM for the detection of four Anti-Patterns (Blob, Spaghetti Code, Functional Decomposition, and Swiss Army Knife). Kaur et al.[58] developed a detection technique based on the SVM algorithm using the polynomial kernel, which detects four smells (Long Method, God Class, Data Class, and Feature Envy). The author called this technique Support Vector Machine Code Smell Detection (SVMCSD).

The use of Association Rules algorithms is common in ML, and authors like Palomba et al.[101] present the HIST (Historical Information for Smell deTection) approach, where they extract change history information from version control systems and then use Association Rules to do smells detection. Fu and Shen[45] also use change history information extracted from version control systems, which they combine with Association Rules (using algorithms such as Apriori or FP-growth) for the detection of 3 smells (*Duplicated Code*, *Shotgun Surgery*, and *Divergent Change*). Czibula et al.[29] based on the relational association rule mining, proposed the detection of the defective classes.

Khomh et al.[63, 65] proposed CS detection using Bayesian belief networks (BBN). This approach aims to manage the uncertainty inherent in the detection process.

More recently, detection techniques based on Artificial Neural Network algorithms have been used. Kim [66] used neural network models to detect six code smells (*God Class*, *Large Class*, *Feature Envy*, *Parallel Inheritance Hierarchies*, *Data Class*, *Lazy Class)*, using a dataset consisting of 8 code metrics extracted from twenty Java projects shared on the GitHub repositories for training and testing. Hadj-Kacem and Bouassida[50] and Liu[74] used *Deep Learning* techniques

in CS detection, which allows them to automatically select code features and build neural networks for detection.

Other ML algorithms are less used in smells detection, such as Binary Logistic Regression or Artificial Immune Systems (AIS). Bryton et al.[17] used code metrics to calibrate a Binary Logistic Regression model that estimates the probability of a method being a Long Method. Hassaine et al. proposed using AIS for the detection of 3 design smells (Blob, Functional Decomposition, and Spaghetti Code) [53].

Most studies using the ML approach only use one algorithm. We now present the three most relevant studies that use multiple ML algorithms. Fontana et al.[6, 42] presented two studies where in the first [42] used six different ML algorithms to detect four code smells (Data Class, Large Class, Feature Envy, Long Method). The second study [6] used 16 ML algorithms to detect the same four code smells. In both studies, the results using 10-fold cross-validation to assess the performance of predictive models showed that J48, Random Forest presented the best results, with *Accuracy* values greater than 90%. Nucci et al.[31] replicated the Fontana et al. study [6] with a more realistic dataset configuration and obtained results with lower performance, e.g., the *Accuracy* of all models on average decreased from 96% to 76%.

This thesis presents a tool that can be configured with the ML algorithms you want since it is composed of microservices, and one of the microservices is responsible for managing the ML component.

5.4 Crowdsmelling

5.4.1 Proposed approach

This approach includes a code metrics extractor. For that purpose, we will use the Eclipse Java metamodel (EJMM) defined in [55]. The EJMM was obtained by reverse engineering and composing two Eclipse JDT components: the Eclipse Java Model (EJM) and the Eclipse Abstract Syntax Tree (or AST, for short). The EJM contains several interfaces that provide a vision over a Java project's structure under a tree architecture. The AST, on the other hand, deals with parsed source code. It allows the analysis of a source code file, also represented as a tree, down to each statement and expression that compose the methods of a class. Although the EJM already provides a fairly complete vision of the software's structure (including, for instance, which classes are declared and their attributes and methods), the AST provides the minutia of a software application that can only be found within the code itself. The two components complement each other to create a highly detailed Java metamodel.

The metrics are formalized using Object Constraint Language (OCL) [137] expressions upon the EJMM. We claim that this approach will allow a much better clarification of the influential factors used to build code smells detection algorithms. A further advantage of this metamodel-based approach is that the OCL formalization of the metrics, whatever their scale type is (e.g., ordinal, interval, absolute, or ratio), is executable upon the EJMM instantiation, using an approach similar to that of M2DM (Metamodel Driven Measurement), that was initially proposed in [1] and has, since then, been successfully applied in several contexts, such as [47] or [28].

5.4.2 Proposed architecture for an application using approach

In this section, we will describe the architecture of the *CrowdSmelling* tool with the presentation of its component diagram in UML.

CrowdSmelling's architecture is based on a set of microservices that communicate through a RESTful API (Application Program Interface). It is composed of three nodes (see figure 5.1): i) *Developer Client*, the developer's computer where the Eclipse IDE plugin is installed, ii) *CrowdSmelling Server*, hosted in a cloud sever contains several docker containers, iii) *Researcher Client*, the researcher's computer component responsible for managing the system.

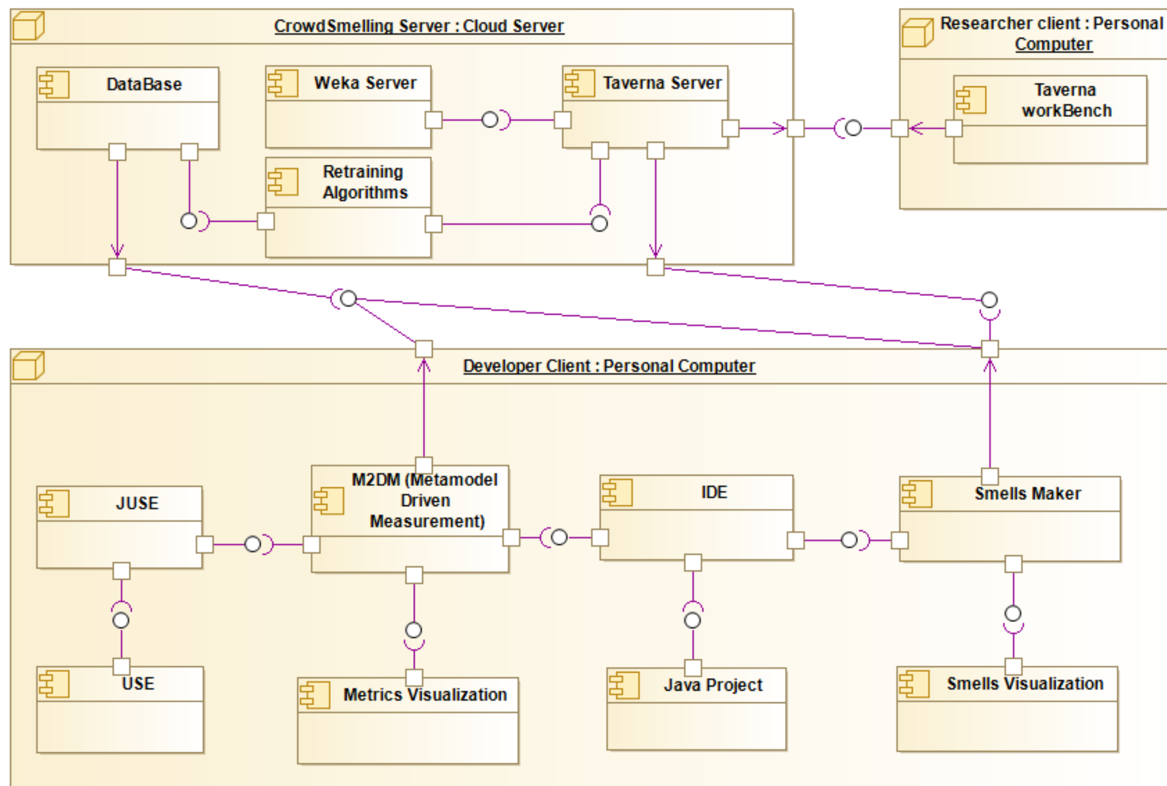


Figure 5.1: Component Diagram

The plugin for the Eclipse IDE, which is installed on the developer's client, consists of two main components, *M2DM (Metamodel Driven Measurement)* and *Smells Maker*. The M2DM (Metamodel Driven Measurement) component is responsible for collecting the metrics, previously defined in OCL, of the active Java project in the IDE's editor. *M2DM* interacts with the *USE (UML-based Specification Environment)*[2] component - USE is an interpreter for a subset of UML and OCL- through the *JUSE* interface. The *Metrics Visualization* component, as the name suggests, serves to perform the visualization of code metrics. The *Smells Maker* component is responsible for showing the localization of the code smells detected by the application and interacts with the programmer, allowing him to express his agreement or disagreement with the detection. Finally, we have the *Smells Visualization* component, responsible for presenting the code smells in different views.

[2]https://sourceforge.net/projects/useocl/

The cloud-hosted *CrowdSmelling* Server node consists of four components, present in the cloud and running on docker containers. The *Database* component is responsible for managing the MySQL database where all the information is stored, namely, code metrics, code smells detection, code smells classification by programmers, list of ML models created, and users. The *Weka Server* component is composed of the JGU WEKA REST Service - RESTful API Webservice to Weka Machine Learning Algorithms [3] - responsible for the machine learning services, i.e., training and testing algorithms and models for code smells detection. The *Taverna Server*[141] component is responsible for managing and executing the workflows for training and classifying code smells through interaction with the Weka Server. The last component of this node is the *Retraining Algorithms*, which will periodically create and test new code smells detection models, using new datasets resulting from programmers' feedback. This component extracts the datasets from the *Database* component and runs the existing training workflow in the *Taverna Server*.

The third node is the *Researcher Client*, whose function is to monitor the system, ensuring that none of the micro-services in the *CrowdSmelling* server fail since we are using a microservices architecture. It also has installed the *Taverna workbench* application that allows us to design the workflows present in the *Taverna Server* and perform the first tests, ensuring that these workflows work.

5.4.3 Application usage scenarios

In this section, we will describe the use cases, followed by a description of Business Process Model (BPMN) processes, and at last, we present a usage example.

5.4.3.1 Use case scenarios

There are two players in the use of this tool, the researchers and the developers (see figure 5.2). The researchers are responsible for two functions: i) the creation and placement in the taverna server (workflow management system) of the training workflows and workflows for the detection of code smells; ii) the monitoring of the system, ensuring that all components are working correctly (we are working with microservices, where some may stop.), visualizing on indicators.

As the primary goal of this tool is to provide developers with code quality improvement, the following three main functionalities are implemented: i) collect and visualize code metrics; ii) perform code smell detection and visualization; iii) perform manual classification of code smells. The code metrics are the basis for detecting code smells, and by analyzing the metrics, the developer can assess the quality of his code. Furthermore, in the code smells classification functionality, the developer will express his opinion regarding the detection presented by the tool, saying whether he agrees or disagrees with the tool's classification.

[3] This application is developed by the Institute of Computer Science at the Johannes Gutenberg University Mainz, inserted in the open risk project.

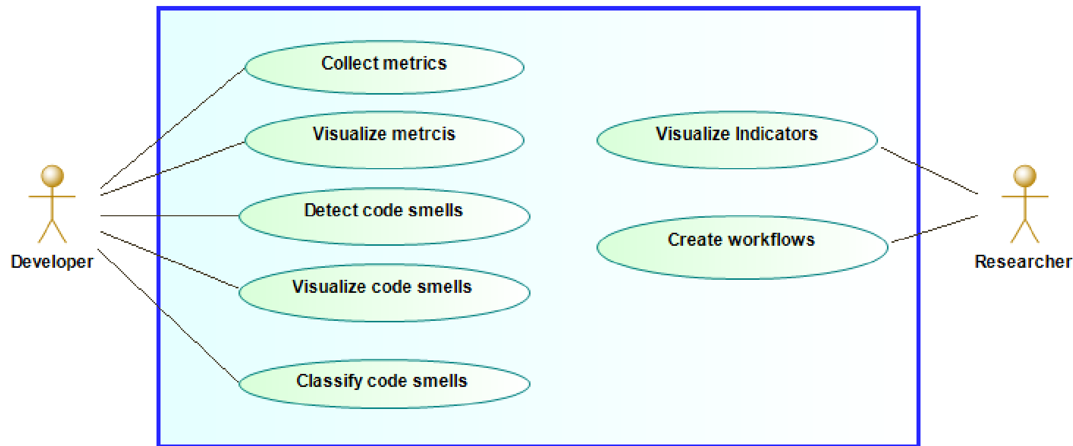


Figure 5.2: Use Case Diagram

5.4.3.2 Scenarios' process

As we are in the presence of a tool with a microservices architecture, the first process to be executed is by the researcher. First, the researcher has to create, through the Taverna workbench application, the workflows for training and classification (see figure 5.3). These workflows are then placed on the Taverna Server, present in the Crowdsmelling Server.

As already mentioned, the CrowdSmelling tool, for the developer, consists of a plugin for the Eclipse IDE, which is run by the developer whenever he/she wants to test the code present in the editor. In figure 5.3 we represent the process of the *CrowdSmelling* approach for the detection and classification of code smells, which consists of the following phases:

i) The first thing the plugin does is collect the code metrics and send them to the code smells detection component of the *CrowdSmelling* Server. The metrics are the input parameters of the ML models in the detection of code smells.

ii) The code smells detection process uses two microservices, Taverna Server and Weka Server, as described in the figure 5.4 in more detail. In the Taverna Server, the workflow for classifying code smells is executed. This workflow interacts with the Weka Server, passing it the code metrics and the ID of the ML model that will be used in the classification (there are several models, depending on the code smell). Each set of metrics of a method or class (depending on the scope of the code smell being detected) is a case to be classified. After performing the classification, the Weka Server returns a file containing the classification of all the metric sets. In the end, this file is returned by the Taverna Server to the Crowdsmelling plugin.

iii) The set of code metrics and their respective classification returned by the Taverna Server constitutes a dataset that also includes the location of code smells in the code. This dataset will be presented to the programmer and will be stored in the Crowdsmelling server's database.

iv) After presenting the dataset with the code smells detections, the programmer can view the code smells in a specific view for that code smell, choose to view the code metrics, or classify the code smells.

v) The classification of code smells by the developer is one of the most important tasks of this approach. Through this classification, the programmer gives his opinion about what a code smell is. Figure 5 represents the process of code smells classification by the developer. The

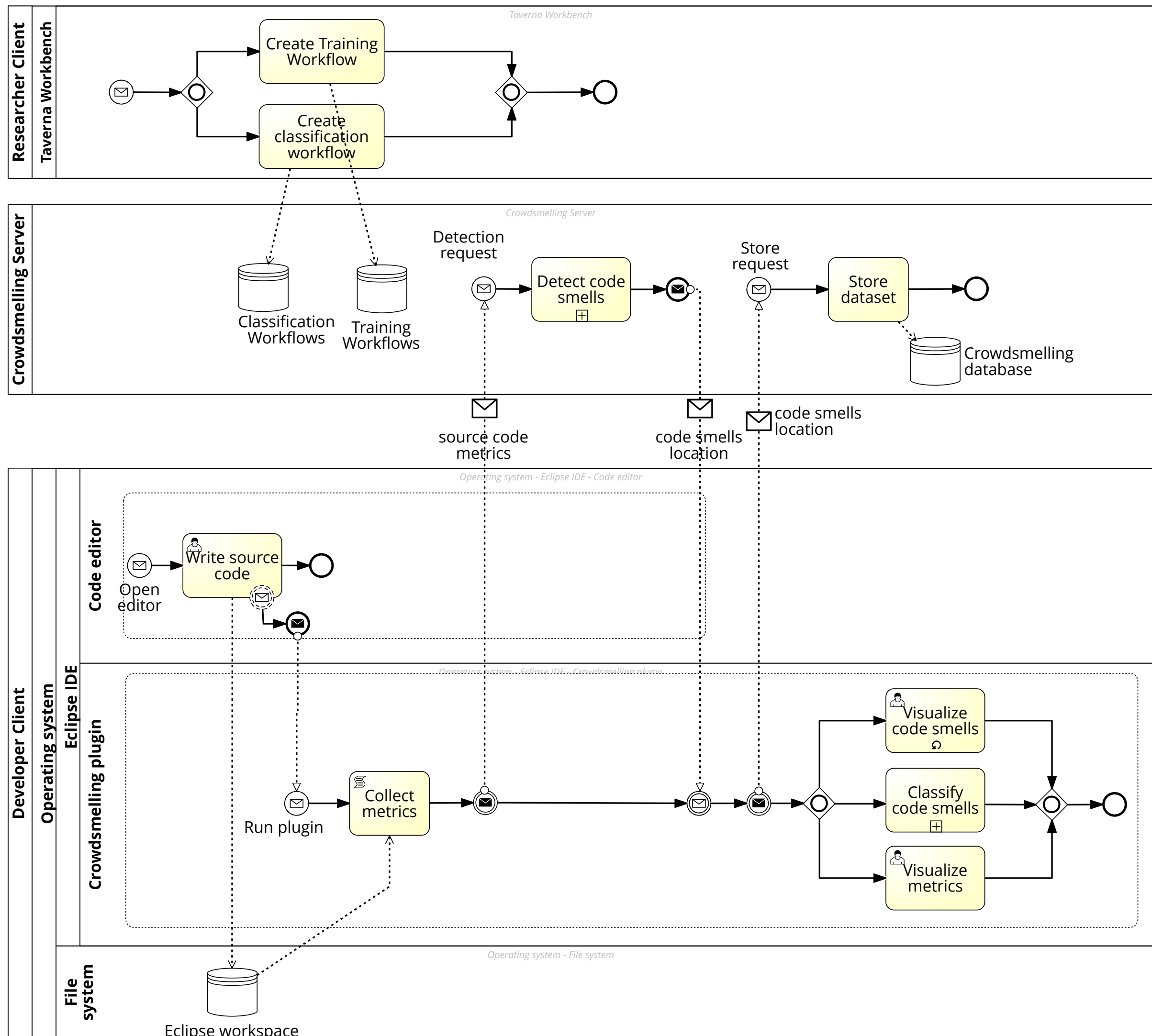


Figure 5.3: The CrowdSmelling approach process

classification displayed is the result of the ML model used, and the developer is now asked to express his opinion by identifying the false positives and false negatives. When the classification of all cases is finished, it is stored in the Crowdsmelling database.

vi) After several developers have given their opinion, manually classifying the code smells, we have new datasets to train the ML algorithms. As a result of this training, new ML models will be created that are better adapted to reality, thus continuously evolving the detection process. This process is performed automatically by the *Retraining Algorithms* component present in the Crowdsmelling Server and is represented in Figure 5.4. Whenever there is a substantial change in the datasets, the process is triggered by calling the training workflow existing in the Taverna Server. This workflow will interact with the Weka Server creating new models that are stored in the Weka Server database. In the end, the workflow returns the model ID that will be saved in the Crowdsmelling database.

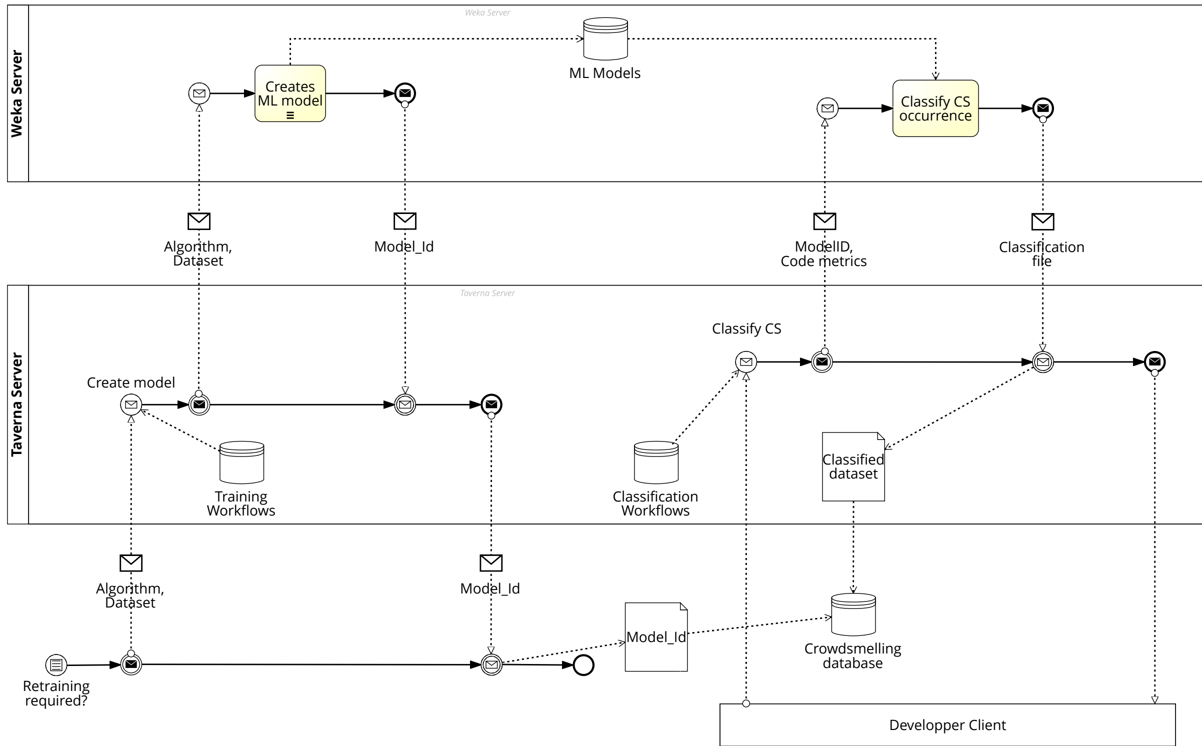


Figure 5.4: The Code smells detection process

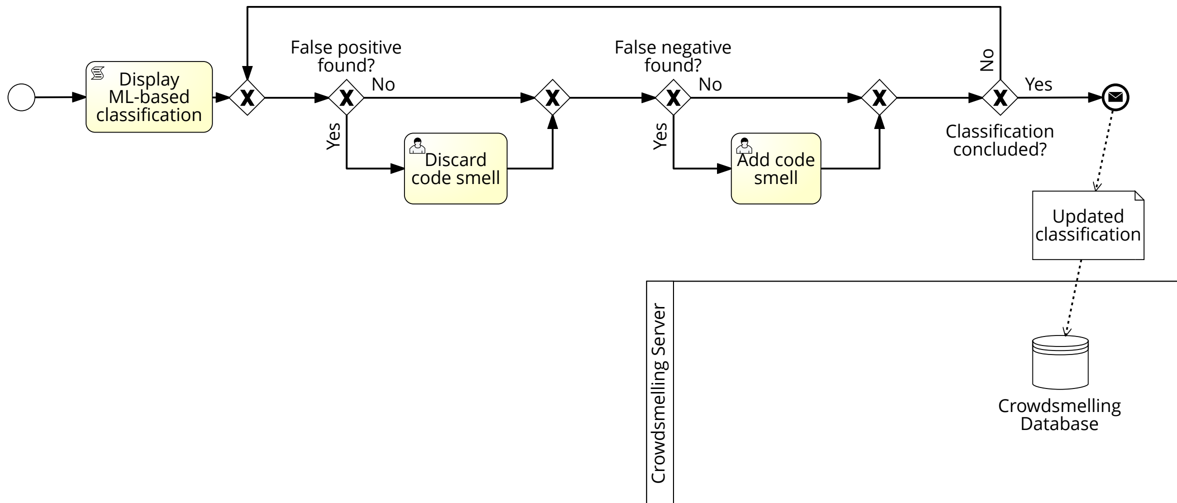


Figure 5.5: The code smells classification process

5.4.3.3 An example of the Crowdsmelling approach usage

In chapter 3 we present an example of using the Crowdsmelling approach, although the process is manual, with the tool proposed in this thesis, we can automate the whole process, starting with extracting the metrics in OCL by navigating the Java Metamodel shown in appendix D.

An example of the calculation of the cyclomatic complexity (*CYCLO*) and lines of code (LOC) metrics:

OCL expression to calculate the cyclomatic complexity (LOC) metric

CYCLO() : Integer=1+self.getAllStatements()→select(s|not(s.oclIsKindOf(ReturnStatement))). conditionalOperatorCount→excluding(oclUndefined(Integer))→sum+self.getAllStatements()→ select(s|s.oclIsKindOf (CatchClause) or s.oclIsKindOf(DoStatement) or s.oclIsKindOf(ForStatement) or s.oclIsKindOf(IfStatement) or (s.oclIsKindOf(SwitchCase) and not(s.oclAsType(SwitchCase). isDefault)) or s.oclIsKindOf(WhileStatement))→size

OCL expression to calculate the lines of code (LOC) metric

LOC() : Integer = self.getAllStatements() → size +self.getAllStatements()→ select(oclIsTypeOf(Block))→ size

Since the metrics are defined in OCL, we can create the ones that are important for the quality of the code. Appendix B.1 presents a table with the main code metrics.

In chapter 3 were built a total of 108 models for detecting code smells. All these models were created and evaluated in WEKA (open-source software from Waikato University), and it was concluded that the results obtained were promising, as they were in line with the results obtained by Nucci [31]. The Nucci study uses more realistic datasets, and therefore more similar to ours. With this tool, we are sure that we will considerably improve these results since we will be able to greatly increase the number of models that we will produce and test.

5.5 Summary

Main conclusions. Code smells' negative impact on code quality has been known for several years. In addition, code smells are a major source of technical debt, causing high costs in software maintenance. Although there are several approaches to detecting code smells, the subjectivity of the detection methods has been an impeding factor in mitigating code smells' impact on code. Machine learning-based detection approaches have started to be used more recently and have advantages over other approaches, such as rule-based approaches, because they do not require the definition of thresholds. However, the detection algorithms of ML techniques need to be trained with realistic oracles. As it happens, there are few publicly available oracles, namely realistic oracles, that is, oracles that reflect the reality of what programmers think about code smells.

To mitigate this problem, we proposed *CrowdSmelling* (see chapter 1), a collaborative crowdsourcing approach based on machine learning, where the wisdom of the crowd (of software developers) is used to collectively calibrate code smells detection algorithms. The first results obtained by this approach showed promise (see chapter 3), but they were obtained through a very manual process, which makes it impractical to apply the *CrowdSmelling* approach on a large scale. Thus, it is necessary to have a tool to apply this approach.

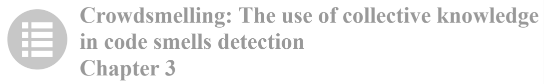
In this chapter, we presented a tool that implements the *CrowdSmelling* approach. This tool is composed of microservices-based architecture and is essentially composed of a plugin for the Eclipse IDE and a cloud-based server hosting a micro-services architecture. The Eclipse IDE plugin is where the developer detects and visualizes the code smells, visualizes the code metrics (collected through the Eclipse Java metamodel obtained by reverse engineering), and classifies the code smells detected by the ML component. Through this classification (agreeing or not with the tool's classification), the developer gives feedback, thus contributing to the enrichment of the datasets. In addition, the ML algorithms are periodically retrained with these new datasets, thus producing models better adapted to reality. This approach can detect any code smells as long as developers can classify examples in the code. The Taverna Server contains the training and classification workflows, and these workflows interact with the machine learning server, in this case, a WEKA Server. There are specific detection models for each code smell, and it is on the Weka server that all these models are created, tested, and used for detection.

Part IV.

Conclusion

PART I: FUNDAMENTALS

PART II: CODE SMELLS DETECTION AND VISUALIZATION



PART III: CROWDSMELLING: A ML-BASED CROWDSOURCING APPROACH FOR CODE SMELLS DETECTION

PART IV: CONCLUSION

This Part concludes this thesis.

Chapter 6.

Conclusion and Future Work

Contents

6.1 Introduction 112
6.2 Thesis Synthesis 112
6.3 Main Contributions 113
6.4 Research Opportunities 114

This chapter summarizes the main contributions of this work, draw opportunities for further research and conclude the thesis.

6.1 Introduction

In software development and maintenance, especially in complex systems, the existence of code smells jeopardizes software quality and hinders several operations, such as code reuse. Code smells are not bugs since they do not prevent a program from functioning, but rather symptoms of software maintainability problems.

The problems start with the definition of a code smell because the absence of a formal definition, making the definition subjective and dependent on developer experience, hampers its detection. Thus, the greater is the experience of the developer, the easiest is code smells detection, as well as the greater complexity of the detected code.

The subjectivity in detection has led to several approaches for detecting code smells. The most widely used approach uses code metrics to create rules. However, these techniques present some problems, such as subjective interpretation, a low agreement between detectors, and threshold dependability. To overcome the aforementioned limitations of code smells detection, researchers recently applied supervised machine learning techniques that can learn from previous datasets without needing any threshold definition. Thus, the main problem of automation is the lack of reliable data to calibrate detection algorithms.

In addition to automatic detection, to alert developers to the presence of smells, the existence of a tool with a visual component is needed to help developers to identify the existence and causes of code smells. However, there is still a significant improvement in the visualization area since the existing solutions require some development, especially for more complex code smells.

This thesis proposes the *Crowdsmelling* approach, a crowdsourcing-based mechanism for obtaining data, allowing dynamically calibrating the Machine Learning algorithms of code smells detection. On the other hand, it is proposed to integrate a visualization tool based on *Smelly Maps* with the detection, thus creating an interactive environment to help programmers in code smells identification.

With this thesis, we hope that our research can create new research topics that contribute to the mitigation of the problems both for developers and their organizations face regarding software quality. The following section summarizes the contributions of each chapter, and provides a roadmap for future work.

6.2 Thesis Synthesis

In chapter 1, we presented the scope, importance of the topic, the research problems and their importance, and the related questions to support the purpose of this dissertation: i) the problem of the subjectivity of defining code smells, ii) the difficulty of detecting code smells and existing approaches, iii) our proposal for detecting and visualizing code smells.

Chapter 2 performed a systematic literature review to characterize the current state of the art in code smells detection and visualization. We concluded that several open issues still exist, such as (i) the definition of code smells, (ii) open-source code smells detection tools are mainly for Java and only detect a small percentage of Fowler's catalog, (iii) there are few publicly available oracles for training ML algorithms, (iv) code smells visualization techniques seem to

have great potential, especially in large systems, to help developers in deciding if they agree with a code smells occurrence suggested by an existing oracle, but there is a need to increase their diversity. Finally, we conducted surveys of the detection and visualization communities to validate our findings.

In order to validate our *CrowdSmelling* approach, we conducted an experiment over three years. In Chapter 3, we presented the results of this experiment. This experiment involved about 100 teams, with an average of 3 elements, which classified the existence of 3 code smells. With this data, we created a set of oracles used to train six ML algorithms. In the end, more than 100 ML models were evaluated to determine the best model to detect each code smell. Good performances were obtained for God Class and Long Method detection, and lower ones for Feature Envy. The results suggest that *CrowdSmelling* is a feasible approach for detecting code smells.

After detecting code smells, we can proceed to their visualization to better understand their side effects and the diagnosis of their cure. In Chapter 4, we propose the concept of *Smelly Maps*. *Smelly Maps* will act as a front-end for the more complex code smells, making it easier to understand their side effects and diagnose their cure. We propose to offer *Smelly Maps* as a set of new views in SourceMiner, which will work in cooperation with *CrowdSmelling*.

In chapter Chapter 5, we presented a tool to implement the *CrowdSmelling* approach. It consists of a plugin for the Eclipse IDE, connected to a microservices architecture, composed of a scientific workflow management system (Taverna 2), an ML algorithm execution platform (Weka), and a database management system that communicate through REST interfaces. Our goal is to obtain real datasets and thus adjust the code smells detection models to the industrial reality, mitigating the problem of detection subjectivity.

Chapter 6 presents a summary of each chapter of the thesis, a summary of the main contributions, and finally, research opportunities in the detection and visualization of code smell.

6.3 Main Contributions

This section presents a summary of the main contributions to quality in software design proposed by this thesis:

- **Crowdsmelling approach for Code smells detection.** Using the concept of Crowdsourcing, we propose a new approach for detecting code smells. This approach uses crowd wisdom to create oracles (a tagged dataset for training detection algorithms) that will train a set of machine learning algorithms, thus producing a set of detection models. The last step is to evaluate which model best detects each code smell. The goal is to mitigate the subjectivity of code smells detection through an automatic, dynamic approach that can detect any code smell.

- ***Smelly Maps* for Code smells visualization.** For a better identification and understanding of code smells, especially in large or more complex systems, we propose the concept of Smelly Map as a complement to detection. *Smelly Maps* are sets of specific views to visualize each type of code smell. The implementation of the *Smelly Maps* is based on the

use of *SourceMiner*, a Multiple Views Interactive Environments (MVIE) implemented as an Eclipse plugin.

- **ML-based crowdsourcing approach for code smells detection.** We present the architecture of a microservices-based code smells detection and visualization tool with an IDE plugin as its front-end and a cloud backend. This tool makes it possible to automate the entire *CrowdSmelling* process, from code metrics extraction, detection, classification, ML model creation to code smells visualization.

- **Lack of publicly available datasets.** One of the problems in using machine learning techniques in code smells detection is that few oracles (a tagged dataset for training detection algorithms) are publicly available. In the context of the experiment we conducted to validate our *CrowdSmelling* approach, we produced a set of 18 Oracles that we made publicly available at GitHub[1] and Zenodo [112], helping to mitigate the scarcity of open-access datasets.

- **Systematic Literature Review (SLR).** We chose to perform an SLR, as this provides a fair evaluation of the current state of the art on code smells detection and visualization, using a trustworthy, rigorous, and auditable methodology.

- **Implications for researchers and practitioners.** Unlike most existing approaches, the *CrowdSmelling* approach does not require the definition of detection rules and thresholds, so it is easy to use without needing to be an expert in code smells. Furthermore, when users validate the detection produced, giving their opinion, they contribute to dynamic learning that is progressively better adapted to the users' reality. Although the results obtained are good (*ROC* of 0.870 for the long Method and 0.896 for the God Class, further validation of the approach in a professional environment is necessary. Furthermore, this approach is not limited to the three code smells studied; it can learn to detect other code smells simply by users classifying them. Thus, it is intended to perform the study for more code smells, but the involvement of researchers and practitioners is needed. In point 3.5.2 - Implications and limitations of the *CrowdSmelling* Approach, point 3.5 - Discussion, is more information about the implication of this approach for researchers and practitioners.

6.4 Research Opportunities

In the future, we intend to continue to develop the *CrowdSmelling* approach by using this tool in an enterprise environment. Our goal is to obtain real datasets and thus adjust the code smells detection models to the industrial reality, mitigating the problem of detection subjectivity. One of the problems with using ML techniques to detect code smells is that there are few publicly available oracles and for few codes smells. We intend to make the oracles produced available to the entire academic community so that ML techniques based on more realistic datasets can be further developed.

[1] https://github.com/dataset-cs-surveys/Crowdsmelling

Other research opportunities that we have detected and that are worth exploring in the future are as follows:

- The *Java* language is dominant, with most open-source tools being for Java. Thus, there is a need to extend the coverage of detection tools to other languages, such as *Phyton*.
- Regarding the coverage of detected code smells, only a small percentage of Fowler's catalog is supported, so it is essential to increase research into other less studied but equally important code smells.
- Studies of code smells in mobile and web environments are still scarce. However, because of the importance of these environments in today's life, we see a wide berth for code smells research in these areas.
- Further validation experiments based on dynamic learning are required to a comprehensive coverage of CS to increase external validity. Since the *CrowdSmelling* approach can be used to detect any code smell, we intend to extend the detection to other CS. In the future, we also plan to apply this new approach in an even more real-world, industrial context, by having companies use the detection tool.
- Visualization of code smells still an area that should be further explored, namely by creating interactive environments that facilitate the interaction between developer and visualization tools. The development environment is sometimes already very overloaded, so it is necessary to create visualization techniques that do not overload the IDE even more. Augmented reality-based techniques can provide an extra set of information to the developer without overloading the development environment.

Bibliography

[1] F. Brito e Abreu. *Using OCL to Formalize Object-Oriented Design Metrics Definitions*. Tech. rep. ES007. Lisboa: INESC, 2001. doi: 10.5281/zenodo.1217095.

[2] F. Brito e Abreu, M. Goulão, and R. Esteves. "Toward the Design Quality Evaluation of Object-Oriented Software Systems." In: *5th International Conference on Software Quality*. American Society for Quality. Austin, Texas, EUA: American Society for Quality, 1995, pp. 44–57. doi: 10.5281/zenodo.1217073.

[3] J. Al Dallal. "Identifying refactoring opportunities in object-oriented code: A systematic literature review." In: *Information and Software Technology* 58 (2015), pp. 231–249. issn: 09505849. doi: 10.1016/j.infsof.2014.08.002.

[4] K. Alkharabsheh, Y. Crespo, E. Manso, and J. A. Taboada. "Software Design Smell Detection: a systematic mapping study." In: *Software Quality Journal* (Oct. 2018). issn: 1573-1367. doi: 10.1007/s11219-018-9424-8.

[5] L Amorim, E Costa, N Antunes, B Fonseca, and M Ribeiro. "Experience report: Evaluating the effectiveness of decision trees for detecting code smells." In: *26th International Symposium on Software Reliability Engineering (ISSRE 2015)*. IEEE, 2015, pp. 261–269. doi: 10.1109/ISSRE.2015.7381819.

[6] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino. "Comparing and experimenting machine learning techniques for code smell detection." In: *Empirical Software Engineering* 21.3 (June 2016), pp. 1143–1191. issn: 15737616. doi: 10.1007/s10664-015-9378-4.

[7] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang. "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis." In: *Information and Software Technology* 108 (2019), pp. 115–138. issn: 0950-5849. doi: 10.1016/j.infsof.2018.12.009.

[8] S. Baltes and C. Treude. "Code Duplication on Stack Overflow." In: *42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. Seoul, South Korea: Association for Computing Machinery, 2020, 13—16. isbn: 9781450371261. doi: 10.1145/3377816.3381744.

[9] G. Bavota and B. Russo. "A large-scale empirical study on self-admitted technical debt." In: *13th International Conference on Mining Software Repositories (MSR)*. Austin, Texas, USA: IEEE, 2016, pp. 315–326.

[10] A. Belikov and V. Belikov. "A citation-based, author- and age-normalized, logarithmic index for evaluation of individual researchers independently of publication." In: *F1000Research* 4.884 (2015). DOI: 10.12688/f1000research.7070.1.

[11] J. Bentzien, I. Muegge, B. Hamner, and D. C. Thompson. "Crowd computing: Using competitive dynamics to develop and refine highly predictive models." In: *Drug Discovery Today* 18.9-10 (2013), pp. 472–478. ISSN: 1359-6446. DOI: 10.1016/j.drudis.2013.01.002.

[12] J. P. Bigham, M. S. Bernstein, and E. Adar. "Human-Computer Interaction and Collective Intelligence." In: *The Collective Intelligence Handbook*. Ed. by T. W. Malone and M. S. Bermstein. CMU, 2014.

[13] M Boussaa, W Kessentini, M Kessentini, S Bechikh, and S Ben Chikha. "Competitive coevolutionary code-smells detection." In: *5th International Symposium on Search-Based Software Engineering (SSBSE)*. Ed. by Ucl, Crest, M. Berner and, and Ibm. Vol. 8084 LNCS. 2013, pp. 50–65. ISBN: 9783642397417. DOI: 10.1007/978-3-642-39742-4_6.

[14] L. Breiman. "Random Forests." In: *Machine Learning* 45.1 (Oct. 2001), 5—32. DOI: 10.1023/A:1010933404324.

[15] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil. "Lessons from applying the systematic literature review process within the software engineering domain." In: *Journal of systems and software* 80.4 (2007), pp. 571–583. DOI: 10.1016/j.jss.2006.07.009.

[16] W. H. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. 1st. USA: John Wiley & Sons, 1998. ISBN: 0471197130.

[17] S. Bryton, F. Brito e Abreu, and M. Monteiro. "Reducing subjectivity in code smells detection: Experimenting with the Long Method." In: *7th International Conference on the Quality of Information and Communications Technology (QUATIC 2010)*. 3. 2010, pp. 337–342. ISBN: 9780769542416. DOI: 10.1109/QUATIC.2010.60.

[18] J. Caldeira, F. Brito e Abreu, J. Cardoso, and J. Pereira dos Reis. "Unveiling process insights from refactoring practices." In: *Computer Standards Interfaces* 81 (2022), p. 103587. ISSN: 0920-5489. DOI: 10.1016/j.csi.2021.103587.

[19] F. L. Caram, B. R. D. O. Rodrigues, A. S. Campanelli, and F. S. Parreiras. "Machine Learning Techniques for Code Smells Detection: A Systematic Mapping Study." In: *International Journal of Software Engineering and Knowledge Engineering* 29.02 (2019), pp. 285–316. DOI: 10.1142/S021819401950013X.

[20] S. K. Card, J. D. Mackinlay, and B. Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, 1999, p. 686. ISBN: 1558605339.

[21] G. d. F. Carneiro, M Silva, L Mara, E Figueiredo, C Sant'Anna, A Garcia, and M Mendonca. "Identifying Code Smells with Multiple Concern Views." In: *Software Engineering (SBES), 2010 Brazilian Symposium on*. 2010, pp. 128–137. DOI: 10.1109/SBES.2010.21.

[22] G. d. F. Carneiro, M. Mendonca, and R. Magnavita. "An experimental platform to characterize software comprehension activities supported by visualization." In: *31st International Conference on Software Engineering (ICSE 2009) - Companion Volume*. IEEE, 2009, pp. 441–442. ISBN: 978-1-4244-3495-4. DOI: 10.1109/ICSE-COMPANION.2009.5071052.

[23] G. D. F. Carneiro, M. G. de Mendonça, and Neto. "SourceMiner: Towards an Extensible Multi-perspective Software Visualization Environment." In: *International Conference on Enterprise Information Systems*. Vol. 190. 2014, pp. 242–263. ISBN: 978-3-319-09491-5. DOI: 10.1007/978-3-319-09492-2.

[24] J. C. Carver. "Towards reporting guidelines for experimental replications: A proposal." In: *1st international workshop on replication in empirical software engineering (RESER'10)*. 2010, pp. 1–4.

[25] J. C. Carver, N. Juristo, M. T. Baldassarre, and S. Vegas. "Replications of software engineering experiments." In: *Empirical Software Engineering* 19.2 (2014), pp. 267–276. ISSN: 15737616. DOI: 10.1007/s10664-013-9290-8.

[26] L. Chen and M. A. Babar. "A systematic review of evaluation of variability management approaches in software product lines." In: *Information and Software Technology* 53.4 (2011), pp. 344–362. DOI: 10.1016/j.infsof.2010.12.006.

[27] Z Chen, L Chen, W Ma, and B Xu. "Detecting Code Smells in Python Programs." In: *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*. 2016, pp. 18–23. DOI: 10.1109/SATE.2016.10.

[28] A. Correia and F. Brito e Abreu. "Enhancing the Correctness of BPMN Models." In: *Improving Organizational Effectiveness with Enterprise Information Systems*. Ed. by J. Varajão, M. M. Cruz-Cunha, and R. Martinho. Advances in Business Information Systems and Analytics (ABISA) Book Series. Hershey, PA, USA: IGI-Global, 2015, pp. 241–261. DOI: 10.4018/978-1-4666-8368-6.Ch015.

[29] G. Czibula, Z. Marian, and I. G. Czibula. "Detecting software design defects using relational association rule mining." In: *Knowledge and Information Systems* 42.3 (2014), pp. 545–577. ISSN: 02193116. DOI: 10.1007/s10115-013-0721-z.

[30] R. de Mello, R. Oliveira, L. Sousa, and A. Garcia. "Towards Effective Teams for the Identification of Code Smells." In: *10th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE 2017)*. IEEE, 2017, pp. 62–65. DOI: 10.1109/CHASE.2017.11.

[31] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. "Detecting code smells using machine learning techniques: Are we there yet?" In: *25th International Conference on Software Analysis, Evolution and Reengineering (SANER 2018)*. IEEE, 2018, pp. 612–621. DOI: 10.1109/SANER.2018.8330266.

[32] S. Diehl. *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

[33] T. Dyba and T. Dingsøyr. "Empirical studies of agile software development: A systematic review." In: *Information and Software Technology* 50.9-10 (2008), pp. 833–859. ISSN: 09505849. DOI: 10.1016/j.infsof.2008.01.006.

[34] E van Emden and L Moonen. "Java quality assurance by detecting code smells." In: *Ninth Working Conference on Reverse Engineering (WCRE'2002)*. 2002, pp. 97–106. ISBN: 1095-1350 VO -. DOI: 10.1109/WCRE.2002.1173068.

[35] A. M. Fard and A Mesbah. "JSNOSE: Detecting JavaScript Code Smells." In: *13th International Working Conference on Source Code Analysis and Manipulation (SCAM 2013)*. IEEE, 2013, pp. 116–125. DOI: 10.1109/SCAM.2013.6648192.

[36] R. Feldt, T. Zimmermann, G. R. Bergersen, D. Falessi, A. Jedlitschka, N. Juristo, J. Münch, M. Oivo, P. Runeson, M. Shepperd, D. I. Sjøberg, and B. Turhan. "Four commentaries on the use of students and professionals in empirical software engineering experiments." In: *Empirical Software Engineering* 23.6 (2018), pp. 3801–3820. ISSN: 15737616. DOI: 10.1007/s10664-018-9655-0.

[37] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo. "A review-based comparative study of bad smell detection tools." In: *20th International Conference on Evaluation and Assessment in Software Engineering (EASE 2016)*. Limerick, Ireland: ACM, 2016. DOI: 10.1145/2915970.2915984.

[38] J. L. Fleiss, B. Levin, and M. C. Paik. *Statistical Methods for Rates and Proportions.* Third. John Wiley & Sons, 2013, p. 723. ISBN: 9781118625613.

[39] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou. "JDeodorant: Identification and Removal of Feature Envy Bad Smells." In: *International Conference on Software Maintenance (ICSM 2007)*. IEEE, Oct. 2007, pp. 519–520. DOI: 10.1109/ICSM.2007.4362679.

[40] F. A. Fontana, P. Braione, and M. Zanoni. "Automatic detection of bad smells in code: An experimental assessment." In: *Journal of Object Technology* 11.2 (2012). DOI: 10.5381/jot.2012.11.2.a5..

[41] F. A. Fontana, M. Mangiacavalli, D. Pochiero, and M. Zanoni. "On experimenting refactoring tools to remove code smells." In: *Scientific Workshop Proceedings of the XP2015 Conference*. New York, New York, USA: ACM Press, May 2015, pp. 1–8. ISBN: 9781450334099. DOI: 10.1145/2764979.2764986.

[42] F. A. Fontana, M. Zanoni, A. Marino, and M. V. M??ntyl?? "Code smell detection: Towards a machine learning-based approach." In: *International Conference on Software Maintenance (ICSM 2013)*. IEEE, 2013, pp. 396–399. ISBN: 978-0-7695-4981-1. DOI: 10.1109/ICSM.2013.56.

[43] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., July 1999. ISBN: 0-201-48567-2.

[44] Y. Freund and R. E. Schapire. "Experiments with a New Boosting Algorithm." In: *Thirteenth International Conference on International Conference on Machine Learning (ICML'96)*. Bari, Italy: Morgan Kaufmann Publishers Inc., 1996, 148—156. ISBN: 1558604197.

[45] S. Fu and B. Shen. "Code Bad Smell Detection through Evolutionary Data Mining." In: *International Symposium on Empirical Software Engineering and Measurement (ESEM 2015)*. IEEE, 2015, pp. 1–9. ISBN: 1949-3770 VO -. DOI: 10.1109/ESEM.2015.7321194.

[46] T. Gerlitz, Q. M. Tran, and C. Dziobek. "Detection and Handling of Model Smells for MATLAB/Simulink models." In: *Modelling in Automotive Software Engineering (MASE) MODELS'2015*. Vol. 1487. CEUR Workshop Proceedings, 2015, pp. 13–22.

[47] M. Goulão and F. Brito e Abreu. "Formal definition of metrics upon the CORBA component model." In: *First International Conference on the Quality of Software Architectures (QoSA'2005)*. Vol. 3712. Lecture Notes in Computer Science. Erfurt, Germany: Springer, Sept. 2005, pp. 88–105. DOI: 10.1007/11558569_8.

[48] A. Gupta, B. Suri, V. Kumar, S. Misra, T. Blažauskas, and R. Damaševičius. "Software code smell prediction model using Shannon, Rényi and Tsallis entropies." In: *Entropy* 20.5 (2018), pp. 1–20. ISSN: 10994300. DOI: 10.3390/e20050372.

[49] A. Gupta, B. Suri, and S. Misra. "A Systematic Literature Review: Code Bad Smells in Java Source Code." In: *ICCSA 2017*. Vol. 10409. 2017, pp. 665–682. ISBN: 978-3-319-62406-8. DOI: 10.1007/978-3-319-62407-5.

[50] M. Hadj-Kacem and N. Bouassida. "A Hybrid Approach To Detect Code Smells Using Deep Learning." In: *13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018)*. Funchal, Madeira, Portugal: SCITEPRESS - Science and Technology Publications, Lda, 2018, 137––146. ISBN: 9789897583001. DOI: 10.5220/0006709801370146.

[51] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. "The WEKA Data Mining Software: An Update." In: *SIGKDD Explorations Newsletter* 11.1 (Nov. 2009), 10––18. ISSN: 1931-0145. DOI: 10.1145/1656274.1656278.

[52] M. Hammad, H. A. Basit, S. Jarzabek, and R. Koschke. "A systematic mapping study of clone visualization." In: *Computer Science Review* 37 (2020), p. 100266. DOI: 10.1016/j.cosrev.2020.100266.

[53] S Hassaine, F Khomh, Y. G. Guéhéneucy, and S Hamel. "IDS: An immune-inspired approach for the detection of software design smells." In: *7th International Conference on the Quality of Information and Communications Technology, QUATIC 2010*. Ed. by I. Porto, Microsoft, Tice.Pt, and Ibm. 2010, pp. 343–348. ISBN: 9780769542416 (ISBN). DOI: 10.1109/QUATIC.2010.61.

[54] W. S. Humphrey. *The Future of Software Engineering: I in The Watts New? Collection: Columns by the SEI's Watts Humphrey*. Tech. rep. CMU/SEI-2009-SR-024. Pitsburgh, USA: Software Engineering Institute Carnegie mellon University, Nov. 2009.

[55] P. Janeiro Coimbra and F. Brito e Abreu. "The Eclipse Java Metamodel - Scaffolding Software Engineering Research on Java Projects with MDE Techniques." In: *2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD'2014)*. Lisbon, Portugal, 2014, pp. 392–399. DOI: 10.5220/0004715303920399.

[56] G. John and P. Langley. "Estimating Continuous Distributions in Bayesian Classifiers." In: *Eleventh Conference on Uncertainty in Artificial Intelligence (UAI'95)*. San Mateo: Morgan Kaufmann, 1995, pp. 338–345. DOI: 10.5555/2074158.2074196.

[57] A. Kaur. "A Systematic Literature Review on Empirical Analysis of the Relationship Between Code Smells and Software Quality Attributes." In: *Archives of Computational Methods in Engineering* (2019). DOI: 10.1007/s11831-019-09348-6.

[58] A. Kaur, S. Jain, and S. Goel. "A Support Vector Machine Based Approach for Code Smell Detection." In: *2017 International Conference on Machine Learning and Data Science (MLDS)*. 2017, pp. 9–14. DOI: 10.1109/MLDS.2017.8.

[59] A. Kaur, S. Jain, and S. Goel. "SP-J48: a novel optimization and machine-learning-based approach for solving complex problems: special application in software engineering for detecting code smells." In: *Neural Computing and Applications* 32 (June 2020). DOI: 10.1007/s00521-019-04175-z.

[60] A. Kaur, S. Jain, S. Goel, and G. Dhiman. "A Review on Machine-Learning Based Code Smell Detection Techniques in Object-Oriented Software System(s)." In: *Recent Advances in Electrical and Electronic Engineering* (Sept. 2020). DOI: 10.2174/2352096513999200922125839.

[61] M. Kessentini and A. Ouni. "Detecting Android Smells Using Multi-objective Genetic Programming." In: *4th International Conference on Mobile Software Engineering and Systems (MOBILESoft'17)*. Piscataway, NJ, USA: IEEE, May 2017, pp. 122–132. ISBN: 978-1-5386-2669-6. DOI: 10.1109/MOBILESoft.2017.29.

[62] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni. "A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection." In: *Transactions on Software Engineering* 40.9 (Sept. 2014), pp. 841–861. ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2331057.

[63] F Khomh, S Vaucher, Y. G. Guéhéneuc, and H Sahraoui. "BDTEX: A GQM-based Bayesian approach for the detection of antipatterns." In: *Journal of Systems and Software* 84.4 (2011), pp. 559–572. ISSN: 01641212 (ISSN). DOI: 10.1016/j.jss.2010.11.921.

[64] F. Khomh, M. D. Penta, Y. G. Guéhéneuc, and G. Antoniol. "An exploratory study of the impact of antipatterns on class change- and fault-proneness." In: *Empirical Software Engineering* 17.3 (2012), pp. 243–275. ISSN: 13823256. DOI: 10.1007/s10664-011-9171-y.

[65] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui. "A Bayesian Approach for the Detection of Code and Design Smells." In: *2009 Ninth International Conference on Quality Software*. 2009, pp. 305–314. ISBN: 978-1-4244-5912-4. DOI: 10.1109/QSIC.2009.47.

[66] D. Kim. "Finding bad code smells with neural network models." In: *International Journal of Electrical and Computer Engineering* 7.6 (2017), pp. 3613–3621. DOI: 10.11591/ijece.v7i6.pp3613-3621.

[67] B. Kitchenham. "The Role of Replications in Empirical Software Engineering–a Word of Warning." In: *Empirical Software Engineering* 13.2 (Apr. 2008), pp. 219–221. ISSN: 1382-3256. DOI: 10.1007/s10664-008-9061-0.

[68] B. Kitchenham and S. Charters. *Guidelines for performing systematic literature reviews in software engineering*. Tech. rep. EBSE Ver. 2.3. Keele University and Durham University, 2007, pp. 1–57.

[69] J. Kreimer. "Adaptive Detection of Design Flaws." In: *Electronic Notes in Theoretical Computer Science* 141.4 (2005). Fifth Workshop on Language Descriptions, Tools, and Applications (LDTA 2005), pp. 117–136. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2005.02.059.

[70] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc. "Code smells and refactoring: A tertiary systematic review of challenges and observations." In: *Journal of Systems and Software* 167 (2020), p. 110610. ISSN: 0164-1212. DOI: 10.1016/j.jss.2020.110610.

[71] J. R. Landis and G. G. Koch. "The Measurement of Observer Agreement for Categorical Data." In: *Biometrics* 33.1 (Mar. 1977), pp. 159–174. ISSN: 0006341X. DOI: 10.2307/2529310.

[72] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Vol. 1. Springer, 2006, p. 213. ISBN: 9788578110796. DOI: 10.1017/CBO9781107415324.004. eprint: arXiv:1011.1669v3.

[73] J. M. Leimeister. "Collective Intelligence." In: *Business Information Systems Engineering* 2.4 (2010), pp. 245–248. DOI: 10.1007/s12599-010-0114-8.

[74] H. Liu, Z. Xu, and Y. Zou. "Deep Learning Based Feature Envy Detection." In: *33rd International Conference on Automated Software Engineering (ASE 2018)*. Montpellier, France: Association for Computing Machinery, 2018, 385—396. ISBN: 9781450359375. DOI: 10.1145/3238147.3238166.

[75] R. Mahouachi, M. Kessentini, and K. Ghedira. "A new design defects classification: Marrying detection and correction." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 7212 LNCS. 2012, pp. 455–470. ISBN: 9783642288715. DOI: 10.1007/978-3-642-28872-2_31.

[76] A. Maiga, N. Ali, N. Bhattacharya, A. Saban, and E. Aimeur. "SMURF : A SVM-based Incremental Anti-pattern Detection Approach." In: *19th Working Conference on Reverse Engineering (WCRE 2012)*. IEEE, 2012, pp. 466–475. DOI: 10.1109/WCRE.2012.56.

[77] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-g. Guéhéneuc, G. Antoniol, and E. Aimeur. "Support Vector Machine for Anti-Pattern Detection." In: *27th International Conference on Automated Software Engineering (ASE 2012)*. Association for Computing Machinery, 2012, pp. 278–281. DOI: 10.1145/2351676.2351723.

[78] U. Mansoor, M. Kessentini, B. R. Maxim, and K. Deb. "Multi-objective code-smells detection using good and bad design examples." In: *Software Quality Journal* 25.2 (2017), pp. 529–552. ISSN: 15731367. DOI: 10.1007/s11219-016-9309-7.

[79] M. Mantyla, J. Vanhanen, and C. Lassenius. "A taxonomy and an initial empirical study of bad smells in code." In: *International Conference on Software Maintenance (ICSM 2003)*. IEEE, 2003, pp. 381–384. DOI: 10.1109/ICSM.2003.1235447.

[80] M. Mantyla, J. Vanhanen, and C. Lassenius. "Bad smells - humans as code critics." In: *20th International Conference on Software Maintenance (ICSM 2004)*. IEEE, 2004, pp. 399–408. ISBN: 0-7695-2213-0. DOI: 10.1109/ICSM.2004.1357825.

[81] C. Marinescu, R. Marinescu, P. F. Mihancea, and R. Wettel. "iplasma: An integrated platform for quality assessment of object-oriented design." In: *International Conference on Software Maintenance (ICSM 2005) - Industrial and Tool Volume*. IEEE, 2005, pp. 77–80.

[82] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. 1st Edition. Prentice Hall, 2002.

[83] R. Mazza. *Introduction to Information Visualization*. 1st ed. Springer, 2009. ISBN: 1848002181.

[84] M. L. McHugh. "Interrater reliability : the kappa statistic." In: *Biochemica Medica* 22.3 (2012), pp. 276–282.

[85] L. Merino, M. Ghafari, C. Anslow, and O. Nierstrasz. "A systematic literature review of software visualization evaluation." In: *Journal of Systems and Software* 144 (2018), pp. 165–180. ISSN: 0164-1212. DOI: 10.1016/j.jss.2018.06.027.

[86] M. W. Mkaouer. "Interactive Code Smells Detection: An Initial Investigation." In: *Search Based Software Engineering*. Ed. by F. Sarro and K. Deb. Vol. 9962. Lecture Notes in Computer Science. NSF; CREST; Ford; Springer; Univ Michigan; Univ Michigan, Coll Engn & Comp Sci. Raleigh, NC, USA: Springer International Publishing, 2016, pp. 281–287. ISBN: 978-3-319-47106-8. DOI: 10.1007/978-3-319-47106-8_24.

[87] N Moha, Y. G. Guéhéneuc, L Duchien, and A. F. Le Meur. "DECOR: A method for the specification and detection of code and design smells." In: *Transactions on Software Engineering* 36.1 (2010), pp. 20–36. ISSN: 00985589 (ISSN). DOI: 10.1109/TSE.2009.50.

[88] N Moha, Y. G. Guéhéneuc, A. F. Le Meur, L Duchien, and A Tiberghien. "From a domain analysis to the specification and detection of code and design smells." In: *Formal Aspects of Computing* 22.3-4 (2010), pp. 345–361. ISSN: 09345043 (ISSN). DOI: 10.1007/s00165-009-0115-x.

[89] M. Monperrus, M. Bruch, and M. Mezini. "Detecting Missing Method Calls in Object-Oriented Software." In: *24th European Conference on Object-Oriented Programming (ECOOP 2010)*. Ed. by T. D'Hondt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 2–25. DOI: 10.5555/1883978.1883982.

[90] E. Murphy-Hill, T. Barik, and a. P. Black. "Interactive ambient visualizations for soft advice." In: *Information Visualization* 12.2 (2013), pp. 107–132. ISSN: 1473-8716. DOI: 10.1177/1473871612469020.

[91] G. Noblit and R. Hare. *Meta-Ethnography: Synthesizing Qualitative Studies*. Qualitative Research Methods. SAGE, 1988. ISBN: 9781506349824.

[92] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjøberg. "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems." In: *International Conference on Software Maintenance (ICSM 2010)*. IEEE, 2010, pp. 1–10. DOI: 10.1109/ICSM.2010.5609564.

[93] R. Oliveira, B. Estácio, A. Garcia, S. Marczak, R. Prikladnicki, M. Kalinowski, and C. Lucena. "Identifying Code Smells with Collaborative Practices: A Controlled Experiment." In: *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*. 2016, pp. 61–70. DOI: 10.1109/SBCARS.2016.18.

[94] R. Oliveira, L. Sousa, R. de Mello, N. Valentim, A. Lopes, T. Conte, A. Garcia, E. Oliveira, and C. Lucena. "Collaborative Identification of Code Smells: A Multi-Case Study." In: *39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP 2017)*. IEEE, 2017, pp. 33–42. DOI: 10.1109/ICSE-SEIP.2017.7.

[95] R. Oliveira. "When More Heads Are Better than One? Understanding and Improving Collaborative Identification of Code Smells." In: *38th International Conference on Software Engineering Companion (ICSE-C 2016)*. IEEE, 2016, pp. 879–882. DOI: 10.1145/2889160.2889272.

[96] R. Oliveira, R. de Mello, E. Fernandes, A. Garcia, and C. Lucena. "Collaborative or individual identification of code smells? On the effectiveness of novice and professional developers." In: *Information and Software Technology* 120 (2020), p. 106242. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2019.106242.

[97] F Palomba, G Bavota, M. D. Penta, R Oliveto, D Poshyvanyk, and A. D. Lucia. "Mining Version Histories for Detecting Code Smells." In: *Transactions on Software Engineering* 41.5 (2015), pp. 462–489. ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2372760.

[98] F Palomba, D Di Nucci, A Panichella, A Zaidman, and A De Lucia. "Lightweight detection of Android-specific code smells: The aDoctor project." In: *24th International Conference on Software Analysis, Evolution, and Reengineering (SANER 2017)*. IEEE, Feb. 2017, pp. 487–491. DOI: 10.1109/SANER.2017.7884659.

[99] F. Palomba, D. Di Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, and A. De Lucia. "Landfill: An Open Dataset of Code Smells with Public Evaluation." In: *12th Working Conference on Mining Software Repositories (MSR 2015)*. IEEE, 2015, pp. 482–485. DOI: 10.1109/MSR.2015.69.

[100] F Palomba, A Panichella, A. D. Lucia, R Oliveto, and A Zaidman. "A textual-based technique for Smell Detection." In: *24th International Conference on Program Comprehension (ICPC 2016)*. IEEE, 2016, pp. 1–10. ISBN: VO -. DOI: 10.1109/ICPC.2016.7503704.

[101] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. "Detecting bad smells in source code using change history information." In: *28th International Conference on Automated Software Engineering (ASE 2013)*. IEEE, 2013. ISBN: 9781479902156. DOI: 10.1109/ASE.2013.6693086.

[102] A. J. Paramita and M. Z. Catur Candra. "CODECOD: Crowdsourcing Platform for Code Smell Detection." In: *2018 5th International Conference on Data and Software Engineering (ICoDSE)*. 2018, pp. 1–6. doi: 10.1109/ICODSE.2018.8705923.

[103] F. Pecorelli, D. Di Nucci, C. De Roover, and A. De Lucia. "On the Role of Data Balancing for Machine Learning-Based Code Smell Detection." In: *3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE 2019)*. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 19–24. doi: 10.1145/3340482.3342744.

[104] J. C. Platt. "Fast training of support vector machines using sequential minimal optimization." In: *Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1999, 185––208. doi: 10.5555/299094.299105.

[105] S. Proksch, S. Amann, and M. Mezini. "Towards standardized evaluation of developer-assistance tools." In: *4th International Workshop on Recommendation Systems for Software Engineering (RSSE 2014)*. New York, New York, USA: ACM Press, 2014, pp. 14–18. isbn: 9781450328456. doi: 10.1145/2593822.2593827.

[106] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Elsevier, 2014, p. 302.

[107] F. Rahman and P. Devanbu. "How, and why, process metrics are better." In: *International Conference on Software Engineering (ICSE 2013)*. ICSE '13. San Francisco, CA, USA: IEEE, 2013, 432––441. isbn: 9781467330763. doi: 10.1109/ICSE.2013.6606589.

[108] K. E. Rajakumari and T. Jebarajan. "A novel approach to effective detection and analysis of code clones." In: *Third International Conference on Innovative Computing Technology (INTECH 2013)*. 2013, pp. 287–290. isbn: 978-1-4799-0048-0. doi: 10.1109/INTECH.2013.6653701.

[109] G. Rasool and Z. Arshad. "A review of code smell mining techniques." In: *Journal of Software: Evolution and Process* 27.11 (2015), pp. 867–895. issn: 2047-7473. doi: 10.1002/smr.1737.

[110] D. Rattan, R. Bhatia, and M. Singh. "Software clone detection: A systematic review." In: *Information and Software Technology* 55.7 (2013), pp. 1165–1199. issn: 09505849. doi: 10.1016/j.infsof.2013.01.008.

[111] J. Pereira dos Reis, F. Brito e Abreu, and G. Figueiredo Carneiro. *Dataset on Code Smells Surveys*. July 2020. doi: 10.5281/zenodo.3936663.

[112] J. Pereira dos Reis, F. Brito e Abreu, and G. Figueiredo Carneiro. *Code Smells Dataset (oracles)*. May 2022. doi: 10.5281/zenodo.6555241.

[113] J. Pereira dos Reis, F. Brito e Abreu, and G. de Figueiredo Carneiro. "Code smells detection 2.0: Crowdsmelling and visualization." In: *2017 12th Iberian Conference on Information Systems and Technologies (CISTI)*. June 2017, pp. 1–4. doi: 10.23919/CISTI.2017.7975961.

[114] J. Pereira dos Reis, F. Brito e Abreu, G. de Figueiredo Carneiro, and C. Anslow. "Code Smells Detection and Visualization: A Systematic Literature Review." In: *Archives of Computational Methods in Engineering* (2021). ISSN: 1886-1784. DOI: 10.1007/s11831-021-09566-x.

[115] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning Internal Representations by Error Propagation." In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. Cambridge, MA, USA: MIT Press, 1986, 318—362. ISBN: 026268053X.

[116] F. Sabir, F. Palma, G. Rasool, Y.-G. Guéhéneuc, and N. Moha. "A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems." In: *Software: Practice and Experience* 49.1 (2019), pp. 3–39. DOI: 10.1002/spe.2639.

[117] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb. "Code-Smell Detection as a Bilevel Problem." In: *Transactions on Software Engineering and Methodology* 24.1 (2014). ISSN: 1049331X. DOI: 10.1145/2675067.

[118] J. A. M. Santos, J. B. Rocha-Junior, L. C. L. Prates, R. S. do Nascimento, M. F. Freitas, and M. G. de Mendonça. "A systematic review on the code smell effect." In: *Journal of Systems and Software* 144 (2018), pp. 450–477. ISSN: 0164-1212. DOI: 10.1016/j.jss.2018.07.035.

[119] G. Saranya, H. Khanna Nehemiah, A. Kannan, and V. Nithya. "Model level code smell detection using EGAPSO based on similarity measures." In: *Alexandria Engineering Journal* 57.3 (2018), pp. 1631–1642. DOI: 10.1016/j.aej.2017.07.006.

[120] M. Sharma and R. Padmanaban. *Leveraging the wisdom of the crowd in software testing*. CRC Press, 2014. ISBN: 9781482254495.

[121] F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo. "The Role of Replications in Empirical Software Engineering." In: *Empirical Software Engineering* 13.2 (Apr. 2008), pp. 211–218. ISSN: 1382-3256. DOI: 10.1007/s10664-008-9060-1.

[122] S. Singh and S. Kaur. "A systematic literature review: Refactoring for disclosing code smells in object oriented software." In: *Ain Shams Engineering Journal* (2017). ISSN: 20904479. DOI: 10.1016/j.asej.2017.03.002.

[123] K Sirikul and C Soomlek. "Automated detection of code smells caused by null checking conditions in Java programs." In: *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. 2016, pp. 1–7. ISBN: VO -. DOI: 10.1109/JCSSE.2016.7748884.

[124] E. V. d. P. Sobrinho, A. De Lucia, and M. d. A. Maia. "A Systematic Literature Review on Bad Smells–5 W's: Which, When, What, Who, Where." In: *IEEE Trans. Softw. Eng.* 47.1 (Jan. 2021), 17–66. ISSN: 0098-5589. DOI: 10.1109/TSE.2018.2880977.

[125] R. Spence. *Information Visualization: Design for Interaction*. 2nd ed. Prentice Hall, 2007, p. 282.

[126] *Stack overflow*. 2008 Accesssed: 2022-05-01.

[127] K.-J. Stol and B. Fitzgerald. “Researching Crowdsourcing Software Development: Perspectives and Concerns.” In: *1st International Workshop on CrowdSourcing in Software Engineering (CSI-SE 2014)*. Association for Computing Machinery, 2014, 7—10. ISBN: 9781450328579. DOI: 10.1145/2593728.2593731.

[128] M. Stone. “Cross-Validatory Choice and Assessment of Statistical Predictions.” In: *Journal of the Royal Statistical Society*. Series B (Methodological) 36.2 (1974), pp. 111–147. ISSN: 0035-9246.

[129] J. Surowiecki. *The Wisdom of Crowds*. Anchor, 2005. ISBN: 0385721706.

[130] A. Tahir, A. Yamashita, S. Licorish, J. Dietrich, and S. Counsell. “Can You Tell Me If It Smells? A Study on How Developers Discuss Code Smells and Anti-Patterns in Stack Overflow.” In: *International Conference on Evaluation and Assessment in Software Engineering (EASE)*. Christchurch, New Zealand: Association for Computing Machinery, 2018, 68—78. ISBN: 9781450364034. DOI: 10.1145/3210459.3210466.

[131] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili. “Detecting defects in object-oriented designs: Using reading techniques to increase software quality.” In: *14th conference on object oriented programming, systems, languages, and applications (OOPSLA)*. New York, NY, USA: ACM Press, 1999, pp. 47–56. DOI: 10.1145/320384.320389.

[132] N Tsantalis, T Chaikalis, and A Chatzigeorgiou. “JDeodorant: Identification and removal of type-checking bad smells.” In: *CSMR 2008 - 12th European Conference on Software Maintenance and Reengineering*. 2008, pp. 329–331. ISBN: 15345351 (ISSN); 9781424421572 (ISBN). DOI: 10.1109/CSMR.2008.4493342.

[133] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. “Ten years of JDeodorant: Lessons learned from the hunt for smells.” In: *25th International Conference on Software Analysis, Evolution and Reengineering (SANER 2018)*. IEEE, 2018, pp. 4–14. DOI: 10.1109/SANER.2018.8330192.

[134] S Vidal, H Vazquez, J. A. Diaz-Pace, C Marcos, A Garcia, and W Oizumi. “JSpIRIT: a flexible tool for the analysis of code smells.” In: *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*. 2015, pp. 1–6. DOI: 10.1109/SCCC.2015.7416572.

[135] W. C. Wake. *Refactoring Workbook*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., 2003.

[136] C. Wang, S. Hirasawa, H. Takizawa, and H. Kobayashi. “Identification and Elimination of Platform-Specific Code Smells in High Performance Computing Applications.” In: *International Journal of Networking and Computing* 5.1 (Jan. 2015), pp. 180–199. ISSN: 2185-2847.

[137] J. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. 1st. Addison-Wesley Object Technology Series, 1998, p. 144.

[138] A. Wasylkowski, A. Zeller, and C. Lindig. "Detecting object usage anomalies." In: *6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Dubrovnik, Croatia: ACM, 2007. DOI: 10.1145/1287624.1287632.

[139] R. Wettel, M. Lanza, and R. Robbes. "Software systems as cities: a controlled experiment." In: *Proceeding of the 33rd international conference on Software engineering - ICSE '11, ACM*. New York, New York, USA: ACM Press, May 2011, pp. 551–560. ISBN: 9781450304450. DOI: 10.1145/1985793.1985868.

[140] C. Wohlin. "Guidelines for snowballing in systematic literature studies and a replication in software engineering." In: *18th International Conference on Evaluation and Assessment in Software Engineering (EASE'14)*. ACM, 2014, pp. 1–10. ISBN: 9781450324762. DOI: 10.1145/2601248.2601268.

[141] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, and C. Goble. "The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud." In: *Nucleic Acids Research* 41.Web Server issue (2013), pp. 557–561. ISSN: 13624962. DOI: 10.1093/nar/gkt328.

[142] A. Yamashita. "Assessing the capability of code smells to support software maintainability assessments: Empirical inquiry and methodological approach." Doctoral dissertation. Faculty of Mathematics and Natural Sciences, University of Oslo, 2012.

[143] A. Yamashita and L. Moonen. "Do code smells reflect important maintainability aspects?" In: *International Conference on Software Maintenance (ICSM 2012)*. IEEE, 2012, pp. 306–315. ISBN: 9781467323123. DOI: 10.1109/ICSM.2012.6405287.

[144] A. Yamashita and L. Moonen. "To what extent can maintenance problems be predicted by code smell detection? - An empirical study." In: *Information and Software Technology* 55.12 (Dec. 2013), pp. 2223–2242. ISSN: 09505849. DOI: 10.1016/j.infsof.2013.08.002.

[145] H. Zhang, M. A. Babar, and P. Tell. "Identifying relevant studies in software engineering." In: *Information and Software Technology* 53.6 (2011), pp. 625–637. DOI: 10.1016/j.infsof.2010.12.010.

[146] M. Zhang, T. Hall, and N. Baddoo. "Code Bad Smells: a review of current knowledge." In: *Journal of Software Maintenance and Evolution* 26.12 (2010), pp. 1172–1192. DOI: 10.1002/smr.521.

[147] M. F. Zibran and C. K. Roy. "IDE-based real-time focused search for near-miss clones." In: *7th Annual ACM Symposium on Applied Computing (SAC 2012)*. Trento, Italy: ACM, 2012. DOI: 10.1145/2245276.2231970.

[148] E Zitzler, L Thiele, M Laumanns, C. M. Fonseca, and V. G. da Fonseca. "Performance Assessment of Multiobjective Optimizers: An Analysis and Review." In: *Transactions on Evolutionary Computation* 7.2 (2003), pp. 117–132. ISSN: 1089-778X. DOI: 10.1109/TEVC.2003.810758.

Appendix A. Systematic Literature Review Materials

A.1 Studies included in the review

ID	Title	Authors	Year	Publish type	Source title
S1	Java quality assurance by detecting code smells	E. van Emden; L. Moonen	2002	Conference	9th Working Conference on Reverse Engineering (WCRE)
S2	Insights into system-wide code duplication	Rieger, M., Ducasse, S., Lanza, M.	2004	Conference	Working Conference on Reverse Engineering, IEEE Computer Society Press
S3	Detection strategies: Metrics-based rules for detecting design flaws	R. Marinescu	2004	Conference	20th International Conference on Software Maintenance (ICSM). IEEE Computer Society Press
S4	Product metrics for automatic identification of "bad smell"design problems in Java source-code	M. J. Munro	2005	Conference	11th IEEE International Software Metrics Symposium (METRICS'05)
S5	Multi-criteria detection of bad smells in code with UTA method	Walter B., Pietrzak B.	2005	Conference	International Conference on Extreme Programming and Agile Processes in Software Engineering (XP)
S6	Adaptive detection of design flaws	Kreimer J.	2005	Conference	Fifth Workshop on Language Descriptions, Tools, and Applications (LDTA)
S7	Visualization-Based Analysis of Quality for Large-Scale Software Systems	G. Langelier, H.A. Sahraoui,; P. Poulin	2005	Conference	20th International Conference on Automated Software Engineering (ASE)
S8	Automatic generation of detection algorithms for design defects	Moha N., Guéhéneuc Y.-G., Leduc P.	2006	Conference	21st IEEE/ACM International Conference on Automated Software Engineering (ASE)
S9	Object - Oriented Metrics in Practice	M. Lanza; R. Marinescu	2006	Book	Springer-Verlag
S10	Detecting Object Usage Anomalies	Andrzej Wasylkowski; Andreas Zeller; Christian Lindig	2007	Conference	6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)
S11	Empirically evaluating the usefulness of software visualization techniques in program comprehension activities	De F. Carneiro G., Orrico A.C.A., De Mendonça Neto M.G.	2007	Conference	VI Jornadas Iberoamericanas de Ingenieria de Software e Ingenieria del Conocimiento (JIISIC)
S12	A Catalogue of Lightweight Visualizations to Support Code Smell Inspection	Chris Parnin; Carsten Gorg; Ogechi Nnadi	2008	Conference	4th ACM Symposium on Software Visualization (SoftVis)
S13	A domain analysis to specify design defects and generate detection algorithms	Moha N., Guéhéneuc Y.-G., Le Meur A.-F., Duchien L.	2008	Conference	International Conference on Fundamental Approaches to Software Engineering (FASE)

S14	JDeodorant: Identification and removal of type-checking bad smells	Tsantalis N., Chaikalis T., Chatzigeorgiou A.	2008	Conference	European Conference on Software Maintenance and Reengineering (CSMR)
S15	Empirical evaluation of clone detection using syntax suffix trees	Raimar Falk, Pierre Frenzel, Rainer Koschke	2008	Journal	Empirical Software Engineering
S16	Visual Detection of Design Anomalies	K. Dhambri, H. Sahraoui,; P. Poulin	2008	Conference	12th European Conference on Software Maintenance and Reengineering (CSMR)
S17	Visually localizing design problems with disharmony maps	Richard Wettel; Michele Lanza	2008	Conference	4th ACM Symposium on Software Visualization (SoftVis)
S18	A Bayesian Approach for the Detection of Code and Design Smells	F. Khomh; S. Vaucher; Y. G. Gueheneuc; H. Sahraoui	2009	Conference	9th International Conference on Quality Software (QSIC)
S19	An Interactive Ambient Visualization for Code Smells	Emerson Murphy-Hill; Andrew P. Black	2010	Conference	5th International Symposium on Software Visualization (SoftVis)
S20	Learning from 6,000 Projects: Lightweight Cross-project Anomaly Detection	Natalie Gruska; Andrzej Wasylkowski; Andreas Zeller	2010	Conference	19th International Symposium on Software Testing and Analysis
S21	Identifying Code Smells with Multiple Concern Views	G. d. F. Carneiro; M. Silva; L. Mara; E. Figueiredo; C. Sant'Anna; A. Garcia; M. Mendonca	2010	Conference	Brazilian Symposium on Software Engineering (SBES)
S22	Reducing Subjectivity in Code Smells Detection: Experimenting with the Long Method	S. Bryton; F. Brito e Abreu; M. Monteiro	2010	Conference	7th International Conference on the Quality of Information and Communications Technology (QUATIC)
S23	DECOR: A method for the specification and detection of code and design smells	Moha N., Guéhéneuc Y.-G., Duchien L., Le Meur A.-F.	2010	Journal	IEEE Transactions on Software Engineering
S24	IDS: An immune-inspired approach for the detection of software design smells	Hassaine S., Khomh F., Guéhéneucy Y.-G., Hamel S.	2010	Conference	7th International Conference on the Quality of Information and Communications Technology (QUATIC)
S25	Detecting Missing Method Calls in Object-Oriented Software	Martin Monperrus Marcel Bruch Mira Mezini	2010	Conference	European Conference on Object-Oriented Programming (ECOOP)

S26	From a domain analysis to the specification and detection of code and design smells	Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, Laurence Duchien, Alban Tiberghien	2010	Journal	Formal Aspects of Computing
S27	BDTEX: A GQM-based Bayesian approach for the detection of antipatterns	Khomh F., Vaucher S., Guéhéneuc Y.-G., Sahraoui H.	2011	Journal	Journal of Systems and Software
S28	IDE-based Real-time Focused Search for Near-miss Clones	Minhaz F. Zibran; Chanchal K. Roy	2012	Conference	27th Annual ACM Symposium on Applied Computing (SAC)
S29	Detecting Bad Smells with Weight Based Distance Metrics Theory	J. Dexun; M. Peijun; S. Xiaohong; W. Tiantian	2012	Conference	2nd International Conference on Instrumentation, Measurement, Computer, Communication and Control (IMCCC)
S30	Analytical learning based on a meta-programming approach for the detection of object-oriented design defects	Mekruksavanich S., Yupapin P.P., Muenchaisri P.	2012	Journal	Information Technology Journal
S31	A New Design Defects Classification: Marrying Detection and Correction	Rim Mahouachi, Marouane Kessentini, Khaled Ghedira	2012	Conference	Fundamental Approaches to Software Engineering
S32	Clones in Logic Programs and How to Detect Them	Céline Dandois, Wim Vanhoof	2012	Conference	Logic-Based Program Synthesis and Transformation
S33	Smurf: A svm-based incremental anti-pattern detection approach	Maiga, A., Ali, N., Bhattacharya, N., Sabane, A., Guéhéneuc, Y-G, & Aimeur, E	2012	Conference	19th Working Conference on Reverse Engineering (WCRE)
S34	Support vector machines for anti- pattern detection	Maiga A, Ali N, Bhattacharya N, Sabané A, Guéhéneuc Y-G, Antoniol G, Aïmeur E	2012	Conference	27th IEEE/ACM International Conference on Automated Software Engineering (ASE)
S35	Detecting Missing Method Calls As Violations of the Majority Rule	Martin Monperrus; Mira Mezini	2013	Journal	ACM Transactions on Software Engineering Methodology
S36	Code Smell Detection: Towards a Machine Learning-Based Approach	F. A. Fontana; M. Zanoni; A. Marino; M. V. Mantyla;	2013	Conference	29th IEEE International Conference on Software Maintenance (ICSM)

S37	Identification of Refused Bequest Code Smells	E. Ligu; A. Chatzigeorgiou; T. Chaikalis; N. Ygeionomakis	2013	Conference	29th IEEE International Conference on Software Maintenance (ICSM)
S38	JSNOSE: Detecting JavaScript Code Smells	A. M. Fard; A. Mesbah	2013	Conference	13th International Working Conference on Source Code Analysis and Manipulation (SCAM)
S39	Interactive ambient visualizations for soft advice	Murphy-Hill E., Barik T., Black A.P.	2013	Journal	Information Visualization
S40	A novel approach to effective detection and analysis of code clones	Rajakumari K.E., Jebarajan T.	2013	Conference	3rd International Conference on Innovative Computing Technology (INTECH)
S41	Competitive coevolutionary code-smells detection	Boussaa M., Kessentini W., Kessentini M., Bechikh S., Ben Chikha S.	2013	Conference	International Symposium on Search Based Software Engineering (SSBSE)
S42	Detecting bad smells in source code using change history information	Palomba F., Bavota G., Di Penta M., Oliveto R., De Lucia A., Poshyvanyk D.	2013	Conference	28th International Conference on Automated Software Engineering (ASE). IEEE/ACM
S43	Code-Smell Detection As a Bilevel Problem	Dilan Sahin; Marouane Kessentini; Slim Bechikh; Kalyanmoy Deb	2014	Journal	ACM Transactions on Software Engineering Methodology
S44	Two level dynamic approach for Feature Envy detection	S. Kumar; J. K. Chhabra	2014	Conference	International Conference on Computer and Communication Technology (ICCCT).
S45	A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection	Kessentini W., Kessentini M., Sahraoui H., Bechikh S., Ouni A.	2014	Journal	IEEE Transactions on Software Engineering
S46	SourceMiner: Towards an Extensible Multi-perspective Software Visualization Environment	Glauco de Figueiredo Carneiro, Manoel Gomes de Mendonça Neto	2014	Conference	International Conference on Enterprise Information Systems (ICEIS)
S47	Including Structural Factors into the Metrics-based Code Smells Detection	Bartosz Walter; Błażej Matuszyk; Francesca Arcelli Fontana	2015	Conference	XP'2015 Workshops
S48	Textual Analysis for Code Smell Detection	Fabio Palomba	2015	Conference	37th International Conference on Software Engineering

S49	Using Developers' Feedback to Improve Code Smell Detection	Mario Hozano; Henrique Ferreira; Italo Silva; Baldoino Fonseca; Evandro Costa	2015	Conference	30th Annual ACM Symposium on Applied Computing (SAC)
S50	Code Bad Smell Detection through Evolutionary Data Mining	S. Fu; B. Shen	2015	Conference	2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)
S51	Mining Version Histories for Detecting Code Smells	F. Palomba; G. Bavota; M. D. Penta; R. Oliveto; D. Poshyvanyk; A. De Lucia	2015	Conference	IEEE Transactions on Software Engineering
S52	Detection and handling of model smells for MATLAB/simulink models	Gerlitz T., Tran Q.M., Dziobek C.	2015	Conference	CEUR Workshop Proceedings
S53	Experience report: Evaluating the effectiveness of decision trees for detecting code smells	Amorim L., Costa E., Antunes N., Fonseca B., Ribeiro M.	2015	Conference	26th International Symposium on Software Reliability Engineering (ISSRE)
S54	Detecting software design defects using relational association rule mining	Gabriela Czibula, Zsuzsanna Marian, Istvan Gergely Czibula	2015	Journal	Knowledge and Information Systems
S55	A Graph-based Approach to Detect Unreachable Methods in Java Software	Simone Romano; Giuseppe Scanniello; Carlo Sartiani; Michele Risi	2016	Conference	31st Annual ACM Symposium on Applied Computing (SAC)
S56	Comparing and experimenting machine learning techniques for code smell detection	Arcelli Fontana F., Mäntylä M.V., Zanoni M., Marino A.	2016	Journal	Empirical Software Engineering
S57	A Lightweight Approach for Detection of Code Smells	Ghulam Rasool, Zeeshan Arshad	2016	Journal	Arabian Journal for Science and Engineering
S58	Multi-objective code-smells detection using good and bad design examples	Usman Mansoor, Marouane Kessentini, Bruce R. Maxim, Kalyanmoy Deb	2016	Journal	Software Quality Journal
S59	Continuous Detection of Design Flaws in Evolving Object-oriented Programs Using Incremental Multi-pattern Matching	Sven Peldszus; Géza Kulcsár; Malte Lochau; Sandro Schulze	2016	Conference	31st IEEE/ACM International Conference on Automated Software Engineering (ASE)

S60	Metric and rule based automated detection of antipatterns in object-oriented software systems	M. T. Aras, Y. E. Selçuk	2016	Conference	2016 7th International Conference on Computer Science and Information Technology (CSIT)
S61	Automated detection of code smells caused by null checking conditions in Java programs	K. Sirikul, C. Soomlek	2016	Conference	2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)
S62	A textual-based technique for Smell Detection	F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, A. Zaidman	2016	Conference	24th International Conference on Program Comprehension (ICPC)
S63	Detecting Code Smells in Python Programs	Z. Chen, L. Chen, W. Ma, B. Xu	2016	Conference	2016 International Conference on Software Analysis; Testing and Evolution (SATE)
S64	Interactive Code Smells Detection: An Initial Investigation	Mkaouer, Mohamed Wiem	2016	Conference	Symposium on Search-Based Software Engineering (SSBSE)
S65	Detecting shotgun surgery bad smell using similarity measure distribution model	Saranya G., Khanna Nehemiah H., Kannan A., Vimala S.	2016	Journal	Asian Journal of Information Technology
S66	Detecting Android Smells Using Multi-objective Genetic Programming	Marouane Kessentini; Ali Ouni	2017	Conference	4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)
S67	Smells Are Sensitive to Developers!: On the Efficiency of (Un)Guided Customized Detection	Mario Hozano; Alessandro Garcia; Nuno Antunes; Baldoino Fonseca; Evandro Costa	2017	Conference	25th International Conference on Program Comprehension (ICPC)
S68	An automated code smell and anti-pattern detection approach	S. Velioglu, Y. E. Selçuk	2017	Conference	2017 IEEE 15th International Conference on Software Engineering Research; Management and Applications (SERA)
S69	Lightweight detection of Android-specific code smells: The aDoctor project	Palomba F., Di Nucci D., Panichella A., Zaidman A., De Lucia A.	2017	Conference	24th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)
S70	On the Use of Smelly Examples to Detect Code Smells in JavaScript	Ian Shoenberger, Mohamed Wiem Mkaouer, Marouane Kessentini	2017	Conference	European Conference on the Applications of Evolutionary Computation (EvoApplications)
S71	A Support Vector Machine Based Approach for Code Smell Detection	A. Kaur; S. Jain; S. Goel	2017	Conference	International Conference on Machine Learning and Data Science (MLDS)

S72	c-JRefRec: Change-based identification of Move Method refactoring opportunities	N. Ujihara; A. Ouni; T. Ishio; K. Inoue	2017	Conference	24th International Conference on Software Analysis, Evolution and Reengineering (SANER)
S73	A Feature Envy Detection Method Based on Dataflow Analysis	W. Chen; C. Liu; B. Li	2018	Conference	42nd Annual Computer Software and Applications Conference (COMPSAC)
S74	A Hybrid Approach To Detect Code Smells using Deep Learning	Hadj-Kacem, M; Bouassida, N	2018	Conference	13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)
S75	Deep Learning Based Feature Envy Detection	Hui Liu and Zhifeng Xu and Yanzhen Zou	2018	Conference	33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)
S76	Detecting Bad Smells in Software Systems with Linked Multivariate Visualizations	H. Mumtaz; F. Beck; D. Weiskopf	2018	Conference	Working Conference on Software Visualization (VisSoft)
S77	Detecting code smells using machine learning techniques: Are we there yet?	D. Di Nucci; F. Palomba; D. A. Tamburri; A. Serebrenik; A. De Lucia	2018	Conference	25th International Conference on Software Analysis, Evolution and Reengineering (SANER)
S78	Exploring the Use of Rapid Type Analysis for Detecting the Dead Method Smell in Java Code	S. Romano; G. Scanniello	2018	Conference	44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)
S79	Model level code smell detection using EGAPSO based on similarity measures	Saranya, G; Nehemiah, HK; Kannan, A; Nithya, V	2018	Journal	Alexandria Engineering Journal
S80	Software Code Smell Prediction Model Using Shannon, Renyi and Tsallis Entropies	Gupta, A; Suri, B; Kumar, V; Misra, S; Blazauskas, T; Damasevicius, R	2018	Journal	Entropy
S81	Towards Feature Envy Design Flaw Detection at Block Level	Ã. Kiss; P. F. Mihancea	2018	Conference	International Conference on Software Maintenance and Evolution (ICSME)
S82	Understanding metric-based detectable smells in Python software: A comparative study	Chen, ZF; Chen, L; Ma, WWY; Zhou, XY; Zhou, YM; Xu, BW	2018	Journal	Information and Software Technology
S83	SP-J48: a novel optimization and machine-learning-based approach for solving complex problems: special application in software engineering for detecting code smells	Amandeep Kaur, Sushma Jain, Shivani Goel	2019	Journal	Neural Computing and Applications

A.2 Studies after applying inclusion and exclusion criteria (phase 3)

ID	Title	Authors	Year	Publish type	Source title
1	Java quality assurance by detecting code smells	E. van Emden; L. Moonen	2002	Conference	9th Working Conference on Reverse Engineering (WCRE)
2	Insights into system-wide code duplication	Rieger, M., Ducasse, S., Lanza, M.	2004	Conference	11th Working Conference on Reverse Engineering (WCRE)
3	Detection strategies: Metrics-based rules for detecting design flaws	R. Marinescu	2004	Conference	20th International Conference on Software Maintenance (ICSM)
4	Product metrics for automatic identification of "bad smell"design problems in Java source-code	M. J. Munro	2005	Conference	11th IEEE International Software Metrics Symposium (METRICS'05)
5	Multi-criteria detection of bad smells in code with UTA method	Walter B., Pietrzak B.	2005	Conference	International Conference on Extreme Programming and Agile Processes in Software Engineering (XP)
6	Adaptive detection of design flaws	Kreimer J.	2005	Conference	Fifth Workshop on Language Descriptions, Tools, and Applications (LDTA)
7	Visualization-Based Analysis of Quality for Large-Scale Software Systems	G. Langelier, H.A. Sahraoui,; P. Poulin	2005	Conference	20th International Conference on Automated Software Engineering (ASE)
8	Automatic generation of detection algorithms for design defects	Moha N., Guéhéneuc Y.-G., Leduc P.	2006	Conference	21st IEEE/ACM International Conference on Automated Software Engineering (ASE)
9	Object - Oriented Metrics in Practice	M. Lanza; R. Marinescu	2006	Book	Springer-Verlag
10	Detecting Object Usage Anomalies	Andrzej Wasylkowski; Andreas Zeller; Christian Lindig	2007	Conference	6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)
11	Using Concept Analysis to Detect Co-change Patterns	Tudor Girba; Stephane Ducasse; Adrian Kuhn; Radu Marinescu; Ratiu Daniel	2007	Conference	9th International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting
12	Empirically evaluating the usefulness of software visualization techniques in program comprehension activities	De F. Carneiro G., Orrico A.C.A., De Mendonça Neto M.G.	2007	Conference	VI Jornadas Iberoamericanas de Ingenieria de Software e Ingenieria del Conocimiento (JIISIC)

13	A Catalogue of Lightweight Visualizations to Support Code Smell Inspection	Chris Parnin; Carsten Gorg; Ogechi Nnadi	2008	Conference	4th ACM Symposium on Software Visualization (SoftVis)
14	A domain analysis to specify design defects and generate detection algorithms	Moha N., Guéhéneuc Y.-G., Le Meur A.-F., Duchien L.	2008	Conference	Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)
15	JDeodorant: Identification and removal of type-checking bad smells	Tsantalis N., Chaikalis T., Chatzigeorgiou A.	2008	Conference	European Conference on Software Maintenance and Reengineering (CSMR)
16	A Survey about the Intent to Use Visual Defect Annotations for Software Models	Jörg Rech, Axel Spriestersbach	2008	Conference	Model Driven Architecture – Foundations and Applications
17	Empirical evaluation of clone detection using syntax suffix trees	Raimar Falk, Pierre Frenzel, Rainer Koschke	2008	Journal	Empirical Software Engineering
18	Visual Detection of Design Anomalies	K. Dhambri, H. Sahraoui, P. Poulin	2008	Conference	12th European Conference on Software Maintenance and Reengineering (CSMR)
19	Detecting bad smells in object oriented design using design change propagation probability matrix	A. Rao; K. Raddy	2008	Conference	International MultiConference of Engineers and Computer Scientists (IMECS)
20	Visually localizing design problems with disharmony maps	Richard Wettel; Michele Lanza	2008	Conference	4th ACM Symposium on Software visualization (SoftVis)
21	A Bayesian Approach for the Detection of Code and Design Smells	F. Khomh; S. Vaucher; Y. G. Gueheneuc; H. Sahraoui	2009	Conference	2009 Ninth International Conference on Quality Software
22	A Flexible Framework for Quality Assurance of Software Artefacts with Applications to Java, UML, and TTCN-3 Test Specifications	J. Nodler; H. Neukirchen; J. Grabowski	2009	Conference	2009 International Conference on Software Testing Verification and Validation (ICST)
23	An Interactive Ambient Visualization for Code Smells	Emerson Murphy-Hill; Andrew P. Black	2010	Conference	5th International Symposium on Software Visualization (SoftVis)
24	Learning from 6,000 Projects: Lightweight Cross-project Anomaly Detection	Natalie Gruska; Andrzej Wasylkowski; Andreas Zeller	2010	Conference	19th International Symposium on Software Testing and Analysis (ISSTA)

25	Identifying Code Smells with Multiple Concern Views	G. d. F. Carneiro; M. Silva; L. Mara; E. Figueiredo; C. Sant'Anna; A. Garcia; M. Mendonca	2010	Conference	Brazilian Symposium on Software Engineering (SBES)
26	Reducing Subjectivity in Code Smells Detection: Experimenting with the Long Method	S. Bryton; F. Brito e Abreu; M. Monteiro	2010	Conference	7th International Conference on the Quality of Information and Communications Technology (QUATIC)
27	DECOR: A method for the specification and detection of code and design smells	Moha N., Guéhéneuc Y.-G., Duchien L., Le Meur A.-F.	2010	Journal	IEEE Transactions on Software Engineering
28	IDS: An immune-inspired approach for the detection of software design smells	Hassaine S., Khomh F., Guéhéneucy Y.-G., Hamel S.	2010	Conference	7th International Conference on the Quality of Information and Communications Technology (QUATIC)
29	Detecting Missing Method Calls in Object-Oriented Software	Martin Monperrus Marcel Bruch Mira Mezini	2010	Conference	European Conference on Object-Oriented Programming (ECOOP)
30	From a domain analysis to the specification and detection of code and design smells	Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, Laurence Duchien, Alban Tiberghien	2010	Journal	Formal Aspects of Computing
31	BDTEX: A GQM-based Bayesian approach for the detection of antipatterns	Khomh F., Vaucher S., Guéhéneuc Y.-G., Sahraoui H.	2011	Journal	Journal of Systems and Software
32	An Approach for Source Code Classification Using Software Metrics and Fuzzy Logic to Improve Code Quality with Refactoring Techniques	Pornchai Lerthathairat, Nakornthip Prompoon	2011	Conference	2nd International Conference on Software Engineering and Computer Systems (ICSECS)
33	IDE-based Real-time Focused Search for Near-miss Clones	Minhaz F. Zibran; Chanchal K. Roy	2012	Conference	27th Annual ACM Symposium on Applied Computing (SAC)
34	Detecting Bad Smells with Weight Based Distance Metrics Theory	J. Dexun; M. Peijun; S. Xiaohong; W. Tiantian	2012	Conference	Second International Conference on Instrumentation, Measurement, Computer, Communication and Control (IMCCC)

35	Analytical learning based on a meta-programming approach for the detection of object-oriented design defects	Mekruksavanich S., Yupapin P.P., Muenchaisri P.	2012	Journal	Information Technology Journal
36	Automatic identification of the anti-patterns using the rule-based approach	Polášek I., Snopko S., Kapustík I.	2012	Conference	10th Jubilee International Symposium on Intelligent Systems and Informatics (SISY)
37	A New Design Defects Classification: Marrying Detection and Correction	Rim Mahouachi, Marouane Kessentini, Khaled Ghedira	2012	Conference	International Conference on Fundamental Approaches to Software Engineering (FASE)
38	Clones in Logic Programs and How to Detect Them	Céline Dandois, Wim Vanhoof	2012	Conference	International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)
39	Smurf: A svm-based incremental anti-pattern detection approach	Maiga, A., Ali, N., Bhattacharya, N., Sabane, A., Guéhéneuc, Y. G., & Aimeur, E	2012	Conference	19th Working Conference on Reverse Engineering (WCRE)
40	Support vector machines for anti- pattern detection	Maiga A, Ali N, Bhattacharya N, Sabané A, Guéhéneuc Y-G, Antoniol G, Aïmeur E	2012	Conference	27th IEEE/ACM International Conference on Automated Software Engineering (ASE)
41	Detecting Missing Method Calls As Violations of the Majority Rule	Martin Monperrus; Mira Mezini	2013	Journal	ACM Transactions on Software Engineering Methodology
42	Code Smell Detection: Towards a Machine Learning-Based Approach	F. A. Fontana; M. Zanoni; A. Marino; M. V. Mantyla;	2013	Conference	29th IEEE International Conference on Software Maintenance (ICSM)
43	Identification of Refused Bequest Code Smells	E. Ligu; A. Chatzigeorgiou; T. Chaikalis; N. Ygeionomakis	2013	Conference	29th IEEE International Conference on Software Maintenance (ICSM)
44	JSNOSE: Detecting JavaScript Code Smells	A. M. Fard; A. Mesbah	2013	Conference	13th International Working Conference on Source Code Analysis and Manipulation (SCAM)
45	Interactive ambient visualizations for soft advice	Murphy-Hill E., Barik T., Black A.P.	2013	Journal	Information Visualization
46	A novel approach to effective detection and analysis of code clones	Rajakumari K.E., Jebarajan T.	2013	Conference	3rd International Conference on Innovative Computing Technology (INTECH)

47	Competitive coevolutionary code-smells detection	Boussaa M., Kessentini W., Kessentini M., Bechikh S., Ben Chikha S.	2013	Conference	Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)
48	Detecting bad smells in source code using change history information	Palomba F., Bavota G., Di Penta M., Oliveto R., De Lucia A., Poshyvanyk D.	2013	Conference	28th IEEE/ACM International Conference on Automated Software Engineering (ASE)
49	Code-Smell Detection As a Bilevel Problem	Dilan Sahin; Marouane Kessentini; Slim Bechikh; Kalyanmoy Deb	2014	Journal	ACM Trans. Softw. Eng. Methodol.
50	Two level dynamic approach for Feature Envy detection	S. Kumar; J. K. Chhabra	2014	Conference	International Conference on Computer and Communication Technology (ICCCT)
51	A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection	Kessentini W., Kessentini M., Sahraoui H., Bechikh S., Ouni A.	2014	Journal	IEEE Transactions on Software Engineering
52	SourceMiner: Towards an Extensible Multi-perspective Software Visualization Environment	Glauco de Figueiredo Carneiro, Manoel Gomes de Mendonça Neto	2014	Conference	International Conference on Enterprise Information Systems (ICEIS)
53	Including Structural Factors into the Metrics-based Code Smells Detection	Bartosz Walter; Błażej Matuszyk; Francesca Arcelli Fontana	2015	Conference	XP'2015 Workshops
54	Textual Analysis for Code Smell Detection	Fabio Palomba	2015	Conference	37th International Conference on Software Engineering (ICSE)
55	Using Developers' Feedback to Improve Code Smell Detection	Mario Hozano; Henrique Ferreira; Italo Silva; Baldoino Fonseca; Evandro Costa	2015	Conference	30th Annual ACM Symposium on Applied Computing (SAC)
56	Code Bad Smell Detection through Evolutionary Data Mining	S. Fu; B. Shen	2015	Conference	2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)
57	JSpIRIT: a flexible tool for the analysis of code smells	S. Vidal; H. Vazquez; J. A. Diaz-Pace; C. Marcos; A. Garcia; W. Oizumi	2015	Conference	34th International Conference of the Chilean Computer Science Society (SCCC)

58	Mining Version Histories for Detecting Code Smells	F. Palomba; G. Bavota; M. D. Penta; R. Oliveto; D. Poshyvanyk; A. De Lucia	2015	Conference	IEEE Transactions on Software Engineering
59	Detection and handling of model smells for MATLAB/simulink models	Gerlitz T., Tran Q.M., Dziobek C.	2015	Conference	CEUR Workshop Proceedings
60	Experience report: Evaluating the effectiveness of decision trees for detecting code smells	Amorim L., Costa E., Antunes N., Fonseca B., Ribeiro M.	2015	Conference	26th International Symposium on Software Reliability Engineering (ISSRE)
61	Detecting software design defects using relational association rule mining	Gabriela Czibula, Zsuzsanna Marian, Istvan Gergely Czibula	2015	Journal	Knowledge and Information Systems
62	A Graph-based Approach to Detect Unreachable Methods in Java Software	Simone Romano; Giuseppe Scanniello; Carlo Sartiani; Michele Risi	2016	Conference	31st Annual ACM Symposium on Applied Computing (SAC)
63	Comparing and experimenting machine learning techniques for code smell detection	Arcelli Fontana F., Mäntylä M.V., Zanoni M., Marino A.	2016	Journal	Empirical Software Engineering
64	A Lightweight Approach for Detection of Code Smells	Ghulam Rasool, Zeeshan Arshad	2016	Journal	Arabian Journal for Science and Engineering
65	Multi-objective code-smells detection using good and bad design examples	Usman Mansoor, Marouane Kessentini, Bruce R. Maxim, Kalyanmoy Deb	2016	Journal	Software Quality Journal
66	Continuous Detection of Design Flaws in Evolving Object-oriented Programs Using Incremental Multi-pattern Matching	Sven Peldszus; Géza Kulcsár; Malte Lochau; Sandro Schulze	2016	Conference	31st IEEE/ACM International Conference on Automated Software Engineering (ASE)
67	Metric and rule based automated detection of antipatterns in object-oriented software systems	M. T. Aras, Y. E. Selçuk	2016	Conference	7th International Conference on Computer Science and Information Technology (CSIT)
68	Automated detection of code smells caused by null checking conditions in Java programs	K. Sirikul, C. Soomlek	2016	Conference	13th International Joint Conference on Computer Science and Software Engineering (JCSSE)
69	A textual-based technique for Smell Detection	F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, A. Zaidman	2016	Conference	24th International Conference on Program Comprehension (ICPC)

70	Detecting Code Smells in Python Programs	Z. Chen, L. Chen, W. Ma, B. Xu	2016	Conference	International Conference on Software Analysis, Testing and Evolution (SATE)
71	DT : a detection tool to automatically detect code smell in software project	Liu, Xinghua; Zhang, Cheng	2016	Conference	4th International Conference on Machinery, Materials and Information Technology Applications
72	Interactive Code Smells Detection: An Initial Investigation	Mkaouer, Mohamed Wiem	2016	Conference	Symposium on Search-Based Software Engineering (SSBSE)
73	Automatic detection of bad smells from code changes	Hammad M., Labadi A.	2016	Journal	International Review on Computers and Software
74	Detecting shotgun surgery bad smell using similarity measure distribution model	Saranya G., Khanna Nehemiah H., Kannan A., Vimala S.	2016	Journal	Asian Journal of Information Technology
75	Detecting Android Smells Using Multi-objective Genetic Programming	Marouane Kessentini; Ali Ouni	2017	Conference	4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)
76	Smells Are Sensitive to Developers!: On the Efficiency of (Un)Guided Customized Detection	Mario Hozano; Alessandro Garcia; Nuno Antunes; Baldoino Fonseca; Evandro Costa	2017	Conference	25th International Conference on Program Comprehension
77	An arc-based approach for visualization of code smells	M. Steinbeck	2017	Conference	24th International Conference on Software Analysis; Evolution and Reengineering (SANER). IEEE
78	An automated code smell and anti-pattern detection approach	S. Velioglu, Y. E. Selçuk	2017	Conference	15th International Conference on Software Engineering Research; Management and Applications (SERA)
79	Lightweight detection of Android-specific code smells: The aDoctor project	Palomba F., Di Nucci D., Panichella A., Zaidman A., De Lucia A.	2017	Conference	24th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)
80	On the Use of Smelly Examples to Detect Code Smells in JavaScript	Ian Shoenberger, Mohamed Wiem Mkaouer, Marouane Kessentini	2017	Conference	European Conference on the Applications of Evolutionary Computation (EvoApplications)
81	A Support Vector Machine Based Approach for Code Smell Detection	A. Kaur; S. Jain; S. Goel	2017	Conference	International Conference on Machine Learning and Data Science (MLDS)

82	An ontology-based approach to analyzing the occurrence of code smells in software	Da Silva Carvalho, L.P., Novais, R., Do Nascimento Salvador, L., De Mendonça Neto, M.G.	2017	Conference	19th International Conference on Enterprise Information Systems (ICEIS)
83	Automatic multiprogramming bad smell detection with refactoring	Verma, A; Kumar, A; Kaur, I	2017	Journal	International Journal of Advanced and Applied Sciences
84	c-JRefRec: Change-based identification of Move Method refactoring opportunities	N. Ujihara; A. Ouni; T. Ishio; K. Inoue	2017	Conference	24th International Conference on Software Analysis, Evolution and Reengineering (SANER)
85	Finding bad code smells with neural network models	Kim, D.K.	2017	Journal	International Journal of Electrical and Computer Engineering
86	Metric based detection of refused bequest code smell	B. M. Merzah; Y. E. SelÃ§uk	2017	Conference	9th International Conference on Computational Intelligence and Communication Networks (CICN)
87	Systematic exhortation of code smell detection using JSmell for Java source code	M. Sangeetha; P. Sengottuvelan	2017	Conference	International Conference on Inventive Systems and Control (ICISC)
88	A Feature Envy Detection Method Based on Dataflow Analysis	W. Chen; C. Liu; B. Li	2018	Conference	42nd Annual Computer Software and Applications Conference (COMPSAC)
89	A Hybrid Approach To Detect Code Smells using Deep Learning	Hadj-Kacem, M; Bouassida, N	2018	Conference	13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)
90	Automatic detection of feature envy using machine learning techniques	Özkalkan, Z., Aydin, K., Tetik, H.Y., Sağlam, R.B.	2018	Conference	12th Turkish National Software Engineering Symposium
91	Code-smells identification by using PSO approach	Ramesh, G., Mallikarjuna Rao, C.	2018	Journal	International Journal of Recent Technology and Engineering
92	Deep Learning Based Feature Envy Detection	Hui Liu and Zhifeng Xu and Yanzhen Zou	2018	Conference	33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)
93	Detecting Bad Smells in Software Systems with Linked Multivariate Visualizations	H. Mumtaz; F. Beck; D. Weiskopf	2018	Conference	Working Conference on Software Visualization (VisSoft)
94	Detecting code smells using machine learning techniques: Are we there yet?	D. Di Nucci; F. Palomba; D. A. Tamburri; A. Serebrenik; A. De Lucia	2018	Conference	25th International Conference on Software Analysis, Evolution and Reengineering (SANER)

95	DT: An Upgraded Detection Tool to Automatically Detect Two Kinds of Code Smell: Duplicated Code and Feature Envy	Xinghua Liu and Cheng Zhang	2018	Conference	International Conference on Geoinformatics and Data Analysis
96	Exploring the Use of Rapid Type Analysis for Detecting the Dead Method Smell in Java Code	S. Romano; G. Scanniello	2018	Conference	2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)
97	Model level code smell detection using EGAPSO based on similarity measures	Saranya, G; Nehemiah, HK; Kannan, A; Nithya, V	2018	Journal	Alexandria Engineering Journal
98	Software Code Smell Prediction Model Using Shannon, Renyi and Tsallis Entropies	Gupta, A; Suri, B; Kumar, V; Misra, S; Blazauskas, T; Damasevicius, R	2018	Journal	Entropy
99	Towards Feature Envy Design Flaw Detection at Block Level	Ã. Kiss; P. F. Mihancea	2018	Conference	International Conference on Software Maintenance and Evolution (ICSME)
100	Understanding metric-based detectable smells in Python software: A comparative study	Chen, ZF; Chen, L; Ma, WWY; Zhou, XY; Zhou, YM; Xu, BW	2018	Journal	Information and Software Technology
101	SP-J48: a novel optimization and machine-learning-based approach for solving complex problems: special application in software engineering for detecting code smells	Amandeep Kaur, Sushma Jain, Shivani Goel	2019	Journal	Neural Computing and Applications
102	Visualizing code bad smells	Hammad, M., Alsofriya, S.	2019	Journal	International Journal of Advanced Computer Science and Applications

A.3 Quality assessment

Study	QC1 Venue Quality	QC2 Data Collected	QC3 Findings	QC4 Recognized Relevance	QC5 Validation	QC6 Replication	QC7 Evaluation	QC8 Visualization	Total
S1	1	1	1	1	0	0	1	1	6
S2	1	1	1	1	0	0	0	1	5
S3	1	1	1	1	1	0	1	0	6
S4	1	0	1	1	0	0	1	0	4
S5	1	1	1	1	0	0	0	0	4
S6	1	0	1	1	1	0	1	0	5
S7	1	0	1	1	0	0	1	1	5
S8	1	1	1	1	1	0	1	0	6
S9	1	1	1	1	0	0	0	1	5
S10	1	1	1	1	0	0	1	0	5
S11	0	1	1	1	0	0	0	0	3
S12	0	1	1	0	0	0	1	1	4
S13	0	1	1	1	0	0	0	1	4
S14	1	1	1	1	1	0	1	0	6
S15	1	1	1	1	0	0	1	0	5
S16	0	0	1	0	0	0	1	1	3
S17	1	1	1	1	1	0	1	0	6
S18	1	1	1	1	0	0	1	1	6
S19	0	1	1	1	0	0	0	0	3
S20	0	1	1	1	0	0	0	1	4
S21	1	1	1	1	1	1	1	0	7
S22	1	0	1	1	0	0	0	0	3
S23	1	0	1	1	0	0	0	1	4
S24	1	0	1	1	1	0	1	0	5
S25	0	1	1	1	0	0	1	1	5
S26	1	0	1	1	0	0	1	0	4
S27	1	1	1	1	1	0	1	0	6
S28	1	1	1	1	1	0	1	0	6
S29	1	1	1	1	0	0	1	0	5
S30	1	1	1	1	1	0	1	0	6
S31	1	1	1	1	1	1	1	0	7
S32	0	0	1	0	0	0	0	0	1
S33	1	1	1	1	1	0	1	0	6
S34	0	1	1	1	0	0	1	0	4
S35	1	1	1	1	1	0	1	0	6
S36	1	0	1	1	0	0	0	0	3
S37	1	1	1	1	1	0	1	0	6
S38	1	1	1	0	0	0	1	0	4
S39	1	0	1	1	1	0	1	0	5
S40	1	1	1	1	1	0	1	0	6
S41	1	1	1	1	0	0	1	0	5
S42	1	1	1	1	0	0	1	0	5
S43	1	1	1	1	0	0	0	0	4
S44	1	1	1	1	1	1	1	0	7
S45	1	0	1	1	0	0	1	1	5
S46	0	1	1	1	0	0	0	1	4
S47	0	1	1	1	1	0	1	0	5

S48	1	1	1	1	1	0	1	0	6
S49	1	1	1	1	1	0	1	0	6
S50	0	1	1	1	1	0	1	0	5
S51	1	1	1	1	1	0	1	0	6
S52	1	1	1	1	0	0	1	1	6
S53	0	1	1	1	1	0	1	0	5
S54	1	1	1	1	1	0	1	0	6
S55	1	1	1	1	1	0	1	0	6
S56	1	1	1	1	1	0	1	0	6
S57	0	0	1	1	0	0	0	0	2
S58	1	1	1	1	1	1	1	0	7
S59	0	1	1	1	0	0	1	0	4
S60	1	1	1	1	1	0	1	0	6
S61	1	1	1	1	1	0	0	0	5
S62	1	1	1	1	1	0	1	0	6
S63	1	1	1	1	1	1	1	0	7
S64	0	1	1	1	0	0	1	1	5
S65	1	1	1	1	1	0	1	0	6
S66	1	1	1	1	1	1	1	0	7
S67	0	1	1	1	1	0	1	0	5
S68	0	1	1	0	1	0	1	0	4
S69	1	1	1	1	1	0	1	0	6
S70	0	1	1	1	0	0	1	0	4
S71	0	0	1	1	1	0	0	0	3
S72	0	1	1	1	1	0	1	0	5
S73	0	1	1	0	0	0	1	0	3
S74	1	1	1	0	1	0	1	0	5
S75	0	1	1	1	1	0	1	0	5
S76	1	1	1	1	1	0	1	0	6
S77	0	0	1	0	0	0	0	1	2
S78	1	1	1	1	1	0	1	0	6
S79	0	1	1	1	0	1	1	0	5
S80	0	1	1	1	1	0	1	0	5
S81	0	1	1	1	1	0	1	0	5
S82	1	1	1	0	0	0	0	0	3
S83	0	0	1	0	0	0	1	0	2
S84	0	1	1	1	1	0	1	0	5
S85	0	1	1	0	0	0	1	0	3
S86	0	1	1	0	0	0	0	0	2
S87	0	0	1	0	0	0	0	0	1
S88	1	1	1	0	1	0	1	0	5
S89	1	1	1	1	1	0	1	0	6
S90	0	1	1	0	0	0	1	0	3
S91	0	0	1	0	0	0	0	0	1
S92	1	1	1	1	1	0	1	0	6
S93	1	1	1	1	0	0	0	1	5
S94	0	1	1	1	1	0	1	0	5
S95	0	1	1	0	1	0	0	0	3
S96	1	1	1	1	1	0	1	0	6
S97	0	1	1	1	1	0	1	0	5
S98	0	1	1	1	0	0	1	0	4
S99	1	1	1	1	1	0	0	1	6

S100	1	1	1	1	0	1	1	0	6
S101	1	1	1	1	1	0	1	0	6
S102	0	0	1	1	0	0	1	0	3
Total	63	83	102	85	54	8	78	18	491

A.4 Description of code smells detected in the studies, according to the authors

Code smell	Description	Reference
Alternative Classes with Different Interface	One class supports different classes, but their interface is different	[43, 70]
AntiSingleton	A class that provides mutable class variables, which consequently could be used as global variables	[64]
God Class (Large Class or Blob)	Class that has many responsibilities and consequently contains many methods and variables. The same Single Responsibility Principle (SRP) also applies in this case	[43, 70]
Brain Class	Class that tends to centralize the functionality of the system and consequently complex. They are therefore assumed to be difficult to understand and maintain. However, unlike God Classes, Brain Classes do not use much data from foreign classes and are slightly more cohesive	[72, 92]
Brain Method	Often a method starts out as a "normal"method, but due to more and more functionality being added gets out of control, making it difficult to understand or maintain. Brain methods tend to centralize the functionality of a class.	[72]
Careless Cleanup	The exception resource can be interrupted by another exception	[48]
Class Data Should Be Private	A class that exposes its fields, thus violating the principle of encapsulation	[64]
Closure Smells	Nested functions declared in JavaScript, are called closures. Closures make it possible to emulate object-oriented notions, such as *public*, *private* members, and *privileged* members. Inner functions have access to the parameters and variables - except for the *this* and *argument* variables - of the functions in which they are nested, even after the outer function has returned. Four smells related to the concept of function closures (long scope chaining, closures in loops, variable name conflict in closures, accessing the this reference in closures)	[35]
Code clone/Duplicated code	Consists of equal or very similar passages in different fragments of the same code base	[43, 70]
Comments	Comments should be used with care as they are generally not required. Whenever it is necessary to insert a comment, it is worth checking if the code cannot be more expressive	[43, 70]
Complex Class	A class that has (at least) one large and complex method, in terms of cyclomatic complexity and LOCs	[64]
Complex Container Comprehension	A container comprehension (including list comprehension, set comprehension, dictionary comprehension, and generator expression) that is too complex	[27]
Complex List Comprehension	A list comprehension that is too complex. List comprehensions in Python provide a concise and efficient way to create new lists. However, when list comprehensions contain complex expressions, they are no longer clear. Apparently, it is hard to analyze control flows of complex list comprehensions	[27]

Coupling between JavaScript, HTML, and CSS	In web applications, HTML is meant for presenting content and structure, CSS for styling, and JavaScript for functional behaviour. It is a well-established programming technique, known as division of concerns, to hold these three entities apart. Unfortunately, JavaScript code is frequently mixed with markup and styling code by web developers, which negatively affects software understanding, maintenance and debugging efforts in web applications.	[35]
Data class	A class that only acts as a data container, without any behaviour. Other classes are typically responsible for manipulating their data, which is the case of Feature Envy,	[43, 70]
Data Clump	Data structures that always appear together, and the entire collection loses its sense when one of the elements is not present.	[43, 70]
Dead Code	Characterized by a variable, attribute, or code fragment that is not used anywhere. Typically it is a result of a change in code with insufficient cleaning	[70, 135]
Delegator	Overuse of delegation or misuse of inheritance	[69]
Dispersed Coupling	Refers to a method which is tied to many operations dispersed among many classes throughout the system	[72]
Divergent Change	A single class needs to be changed for many reasons. This is a strong indication that it is not sufficiently cohesive and must be divided	[43, 70]
Dummy Handler	Dummy handler is only used for viewing the exception but it will not handle the exception	[48]
Empty Catch Block	When the catch block is left blank in the catch statement	[48]
Exception thrown in the finally block	How to handle the exception thrown inside the finally block of another try catch statement	[48]
Excessive Global Variables	Global variables can be accessed in the JavaScript code from anywhere, even if they are defined in different files loaded on the same page. As such, naming conflicts between global variables in different JavaScript source files is common, which affects program dependability and correctness. The higher the number of global variables in the code, the more dependent existing modules are likely to be; and dependency increases errorproneness, and maintainability efforts	[35]
Feature Envy	When a method is more interested in members of other classes than its own, is a clear sign that it is in the wrong class	[43, 70]
Functional Decomposition	A procedural code in a technology that implements the OO paradigm (usually the main function that calls many others), caused by the previous expertise of the developers in a procedural language and little experience in OO	[16, 70]
God Package	A package that is too large. That knows too much or does too much	[82]
Inappropriate Intimacy	A case where two classes are known too, characterizing a high level of coupling	[43, 70]
Incomplete Library Class	The software uses a library that is not complete, and therefore extensions to that library are required	[43, 70]

Instanceof	In Java, the instanceof operator is used to check that an object is an instance of a given class or implements a certain interface. These are considered CS aspects because a concentration of instanceof operators in the same block of code may indicate a place where the introduction of an inheritance hierarchy or the use of method overloading might be a better solution	[34]
Intensive Coupling	Refers to a method that is tied to many other operations located in only a few classes within the system.	[72]
Introduce null object	Repeated null checking conditions are added into the code to prevent the null pointer exception problem. By doing so, the duplications of null checking conditions could have been placed in different locations of the software system	[123]
Large object	An object with too many responsibilities. An object that is doing too much should be refactored. Large objects may be restructured or broken into smaller objects	[35]
Lazy Class	Classes that do not have sufficient responsibilities and therefore should not exist	[43, 70]
Lazy object	An object that does too little. An object that is not doing enough work should be refactored. Lazy objects maybe collapsed or combined into other classes	[35]
Long Base Class List	A class definition with too many base classes. Python supports a limited form of multiple inheritance. If an attribute in Python is not found in the derived class during execution, it is searched recursively in the base classes declared in the base class list in sequence. Too long base class list will limit the speed of interpretive execution	[27]
Long Element Chain	An expression that is accessing an object through a long chain of elements by the bracket operator. Long Element Chain is directly caused by nested arrays. It is unreadable especially when a deep level of array traversing is taking place	[27]
Long Lambda Function	A lambda function that is overly long, in term of the number of its characters	[27]
Long Message Chain	An expression that is accessing an object through a long chain of attributes or methods by the dot operator	[27]
Long Method	Very large method/function and, therefore, difficult to understand, extend and modify. It is very likely that this method has too many responsibilities, hurting one of the principles of a good OO design (SRP: Single Responsibility Principle	[43, 70]
Long Parameter List	Extensive parameter list, which makes it difficult to understand and is usually an indication that the method has too many responsibilities	[43, 70]
Long Scope Chaining	A method or a function that is multiply-nested	[27]
Long Ternary Conditional Expression	A ternary conditional expression ("X if C else Y") that is overly long	[27]
Message Chain	One object accesses another, to then access another object belonging to this second, and so on, causing a high coupling between classes	[43, 70]
Method call sequences	The interplay of multiple methods, though—in particular, whether a specific sequence of method calls is allowed or not—is neither specified nor checked at compile time	[138]

Middle Man	Identified how much a class has almost no logic, as it delegates almost everything to another class	[43, 70]
Misplaced Class	Suggests a class that is in a package that contains other classes not related to it	[100]
Missing method calls	Overlook certain important method calls that are required at particular places in code	[89]
Multiply-Nested Container	A container (including set, list, tuple, dict) that is multiply-nested. It directly produces expressions accessing an object through a long chain of indexed elements	[27]
Nested Callback	A callback is a function passed as an argument to another (parent) function. Using excessive callbacks, however, can result in hard to read and maintain code due to their nested anonymous (and usually asynchronous) nature	[35]
Nested Try Statements	When one or more try statements are contained in the try statement	[48]
Null checking in a string comparison problem	Null checking conditions are usually found in string comparison, particularly in an if statement. This form of defensive programming can be employed to prevent the null pointer exception error. The same null checking statement is repeatedly appeared when the same String object is compared, resulting in a marvellous number of duplicated null checking conditions	[123]
Parallel Inheritance	Existence of two hierarchies of classes that are fully connected, that is, when adding a subclass in one of the hierarchies, it is required that a similar subclass be created in the other	[43, 70]
Primitive Obsession	It represents the situation where primitive types are used in place of light classes	[43, 70]
Promiscuous Package	A package can be considered as promiscuous if it contains classes implementing too many features, making it too hard to understand and maintain	[100]
Refused Bequest	It indicates that a subclass does not use inherited data or behaviors	[43, 70]
Shotgun Surgery	Opposite to Divergent Change, because when it happens a modification, several different classes have to be changed	[43, 70]
Spaghetti Code	Use of classes without structures, long methods without parameters, use of global variables, in addition to not exploiting and preventing the application of OO principles such as inheritance and polymorphism	[16, 70]
Speculative Generality	Code snippets are designed to support future software behavior that is not yet required	[43, 70]
Swiss Army Knife	Exposes the high complexity to meet the predictable needs of a part of the system (usually utility classes with many responsibilities)	[16, 70]
Switch Statement	It is not necessarily smells by definition, but when they are widely used, they are usually a sign of problems, especially when used to identify the behavior of an object based on its type	[43, 70]
Temporary Field	Member-only used in specific situations, and that outside of it has no meaning	[43, 70]

Tradition Breaker	This design disharmony strategy takes its name from the principle that the interface of a class should increase in an evolutionary fashion. This means that a derived class should not break the inherited "tradition" and provide a large set of services which are unrelated to those provided by its base class.	[72]
Type Checking	Type-checking code is introduced in order to select a variation of an algorithm that should be executed, depending on the value of an attribute	[132]
Typecast	Typecasts are used to explicitly convert an object from one class type into another. Many people consider typecasts to be problematic since it is possible to write illegal casting instructions in the source code which cannot be detected during compilation but result in runtime errors	[34]
Unprotected Main	Outer exception will not be handled in the main program; it can only be handled in a subprogram or a function	[48]
Useless Exception Handling	A try...except statement that does little	[27]
Wide Subsystem Interface	A Subsystem Interface consists of classes that are accessible from outside the package they belong to. The flaw refers to the situation where this interface is very wide, which causes a very tight coupling between the package and the rest of the system	[143]

A.5 Frequencies of code smells detected in the studies

Code smell	Nº of studies	% Studies	programming language
God Class (Large Class or Blob)	43	51.8%	Java, C/C++ , C#, Python
Feature Envy	28	33.7%	Java, C/C++ , C#
Long Method	22	26.5%	Java, C/C++ , C#, Python, JavaScript
Data class	18	21.7%	Java, C/C++ , C#
Functional Decomposition	17	20.5%	Java
Spaghetti Code	17	20.5%	Java
Long Parameter List	12	14.5%	Java, C/C++ , C#, Python, JavaScript
Swiss Army Knife	11	13.3%	Java
Refused Bequest	10	12.0%	Java, C/C++ , C#, JavaScript
Shotgun Surgery	10	12.0%	Java, C++ , C#
Code clone/Duplicated code	9	10.8%	Java, C/C++ , C#
Lazy Class	8	9.6%	Java, C++ , C#
Divergent Change	7	8.4%	Java, C#
Dead Code	4	4.8%	Java, C++ , C#
Switch Statement	4	4.8%	Java, C#, JavaScript
Brain Class	3	3.6%	Java, C++
Data Clump	3	3.6%	Java, C/C++ , C#
Long Message Chain	3	3.6%	JavaScript, Python
Misplaced Class	3	3.6%	Java, C++
Parallel Inheritance	3	3.6%	Java, C#
Primitive Obsession	3	3.6%	Java, C/C++ , C#
Speculative Generality	3	3.6%	Java, C#
Temporary Field	3	3.6%	Java, C#
Dispersed Coupling	2	2.4%	Java, C++
Empty Catch Block	2	2.4%	Java, JavaScript
Excessive Global Variables	2	2.4%	JavaScript
Intensive Coupling	2	2.4%	Java, C++
Large object	2	2.4%	JavaScript
Lazy object	2	2.4%	JavaScript
Long Base Class List	2	2.4%	Python
Long Lambda Function	2	2.4%	Python
Long Scope Chaining	2	2.4%	Python
Long Ternary Conditional Expression	2	2.4%	Python
Message Chain	2	2.4%	Java, C/C++ , C#
Middle Man	2	2.4%	Java, C/C++ , C#
Missing method calls	2	2.4%	Java
Alternative Classes with Different Interface	1	1.2%	Java, C#
AntiSingleton	1	1.2%	Java
Brain Method	1	1.2%	Java, C++
Careless Cleanup	1	1.2%	Java
Class Data Should Be Private	1	1.2%	Java
Closure Smells	1	1.2%	JavaScript
Comments	1	1.2%	Java, C#
Complex Class	1	1.2%	Java
Complex Container Comprehension	1	1.2%	Python
Complex List Comprehension	1	1.2%	Python

Coupling between JavaScript, HTML, and CSS	1	1.2%	JavaScript
Delegator	1	1.2%	Java
Dummy Handler	1	1.2%	Java
Exception thrown in the finally block	1	1.2%	Java
God Package	1	1.2%	Java, C++
Inappropriate Intimacy	1	1.2%	Java, C#
Incomplete Library Class	1	1.2%	Java, C#
Instanceof	1	1.2%	Java
Introduce null object	1	1.2%	Java
Long Element Chain	1	1.2%	Python
Method call sequences	1	1.2%	Java
Multiply-Nested Container	1	1.2%	Python
Nested Callback	1	1.2%	JavaScript
Nested Try Statements	1	1.2%	Java
Null checking in a string comparison problem	1	1.2%	Java
Promiscuous Package	1	1.2%	Java
Tradition Breaker	1	1.2%	Java, C++
Type Checking	1	1.2%	Java
Typecast	1	1.2%	Java
Unprotected Main	1	1.2%	Java
Useless Exception Handling	1	1.2%	Python
Wide Subsystem Interface	1	1.2%	Java, C++

[This page has been intentionally left blank]

Appendix

B. Crowdsmelling Materials

B.1 Code metrics

This table presents the metrics used in the study reported in chapter 3, more information about the metrics can be found at [6] or on the web[1].

Metric	Acronym	Scope
Access to Foreign Data	ATFD	Method
Access to Local Data	ATLD	Method
Average Methods Weight	AMW	Class
Average Methods Weight of Not Accessor or Mutator Methods	AMWNAMM	Class
Called Foreign Not Accessor or Mutator Methods	CFNAMM	Class
Called Local Not Accessor or Mutator Methods	CLNAMM	Method
Changing Classes	CC	Class
Changing Methods	CM	Method
Coupling Between Objects Classes	CBO	Class
Coupling Dispersion	CDISP	class
Coupling Intensity	CINT	Method
Cyclomatic Complexity	CYCLO	Method
Depth of Inheritance Tree	DIT	Class
Fanout	FANOUT	Class
Foreign Data Providers	FDP	Method
Lack of Cohesion in Methods	LCOM	Class
Lines of Code	LOC	Method
Lines of Code Without Accessor or Mutator Methods	LOCNAMM	Class
Locality of Attribute Accesses	LAA	Method
Maximum Message Chain Length	MAMCL	Method
Maximum Nesting Level	MAXNESTIN	Method
Mean Message Chain Length	MEMCL	Method
Number of Abstract Methods	NOABM	Class
Number of Accessed Variables	NOAV	Method
Number of Accessor Methods	NOAM	Class
Number of Attributes	NOA	Class
Number of Children	NOC	Class
Number of Classes	NOCS	Package
Number of Constructor Methods	NOCM	Class
Number of Default Attributes	NODA	Class
Number of Default Methods	NODM	Class
Number of Final and Non - Static Attributes	NOFNSA	Class
Number of Final and Non - Static Methods	NOFNSM	Class
Number of Final and Non - Static Methods	NONFNSM	Class
Number of Final and Static Attributes	NOFSA	Class
Number of Final and Static Methods	NOFSM	Class
Number of Final Attributes	NOFA	Class
Number of Final Methods	NOFM	Class
Number of Implemented Interfaces	NOII	Class
Number of Inherited Methods	NIM	Class
Number of Interfaces	NOI	Package
Number of Local Variable	NOLV	Method
Number of Message Chain Statements	NMCS	Method
Number of Methods	NOM	Class
Number of Methods Overridden	NMO	Class

[1] https://essere.disco.unimib.it/machine-learning-for-code-smell-detection/

Number of Non - Accessors Methods	NONAM	Class
Number of Non - Constructor Methods	NONCM	Class
Number of Non - Final and Non - Abstract Methods	NONFNABM	Class
Number of Non - Final and Static Attributes	NONFSA	Class
Number of Non - Final and Static Methods	NONFSM	Class
Number of Not Accessor or Mutator Methods	NOMNAMM	Class
Number of Not Final and Non - Static Attributes	NONFNSA	Class
Number of Packages	NOPK	Project
Number of Parameters	NOP	Method
Number of Private Attributes	NOPVA	Class
Number of Private Methods	NOPM	Class
Number of Protected Attributes	NOPRA	Class
Number of Protected Methods	NOPRM	Class
Number of Public Attributes	NOPA	Class
Number of Public Methods	NOPLM	Class
Number of Static Attributes	NOSA	Class
Number of Static Methods	NOSM	Class
Response for A Class	RFC	Class
Tight Class Cohesion	TCC	Class
Weight of Class	WOC	Class
Weighted Methods Count	WMC	Class
Weighted Methods Count of Not Accessor or Mutator Methods	WMCNAMM	Class

[This page has been intentionally left blank]

Appendix C.

Architectures of the crowdsmelling tool versions

C.1 Version 1 - Eclipse plugin and Azure Machine Learning

The first version of the crowdsmelling tool consisted of a plugin for the Eclipse IDE, which communicated via web services with Microsoft Azure. All the machine learning component was developed in Microsoft Azure Machine Learning Studio.

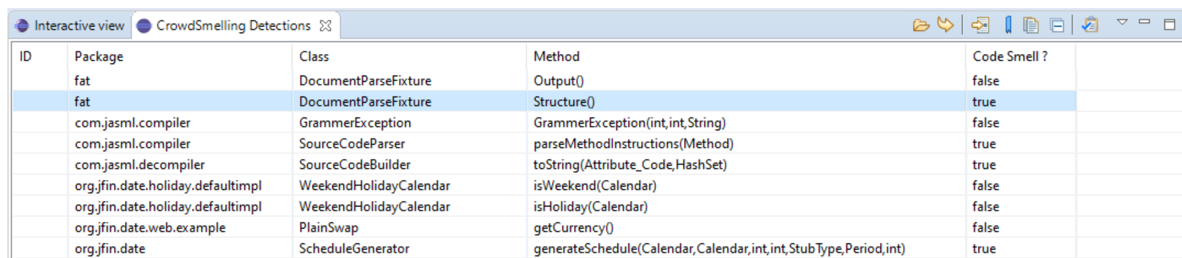
C.1.1 Eclipse IDE plugin

Figure C.1 shows the graphical interface of the plugin. Essentially this plugin consists of 3 components:

i) Code metrics settings. Since this version does not extract the code metrics, it is necessary to import a file containing them. The metrics table in appendix B.1 shows the metrics that are imported, although for the detection of the 3 code smells (*God Class*, *Long Method*, *and Feature Envy*) the models do not use all metrics. For example, based on the J48 decision tree, the model for Long Method detection only uses *Cyclomatic complexity* (*CYCLO*) and the *Number of lines of code* (*LOC*) ;

ii) Code smells detection. After importing the code metrics, code smells can be detected through the existing ML models in Microsoft Azure Machine Learning. The result of the detection is shown in the "Code smell ?"column. The communication is done through Microsoft Azure web services;

iii) Validate and save the validation results. After detecting the code smells, the developers validate it by saying whether they agree or disagree with its result. To validate, they only have to double click on the "Code smell ?"column in the cases where they disagree with the detection. The double click reverses the column's value; if it is true, it becomes false and vice-versa. After validation, all information is stored in the MySQL database in the cloud, communicating through web services.



Interactive view | CrowdSmelling Detections

ID	Package	Class	Method	Code Smell ?
	fat	DocumentParseFixture	Output()	false
	fat	DocumentParseFixture	Structure()	true
	com.jasml.compiler	GrammerException	GrammerException(int,int,String)	false
	com.jasml.compiler	SourceCodeParser	parseMethodInstructions(Method)	true
	com.jasml.decompiler	SourceCodeBuilder	toString(Attribute_Code,HashSet)	true
	org.jfin.date.holiday.defaultimpl	WeekendHolidayCalendar	isWeekend(Calendar)	false
	org.jfin.date.holiday.defaultimpl	WeekendHolidayCalendar	isHoliday(Calendar)	false
	org.jfin.date.web.example	PlainSwap	getCurrency()	false
	org.jfin.date	ScheduleGenerator	generateSchedule(Calendar,Calendar,int,int,StubType,Period,int)	true

Figure C.1: Crowdsmelling Eclipse IDE plugin

C.1.2 Machine Learning Component

Algorithm training, predictions, and model evaluation are performed in Azure's Machine Learning Studio. For each code smell, training workflows are built for the various algorithms, and then the resulting models are evaluated. Finally, the best model is chosen to detect the code smell. Figure C.2 shows an example of the workflow for creating the Feature Envy detection model, and Figure C.3 shows the Feature Envy prediction workflows. The communication of the Eclipse plugin with the ML detection component is done through Microsoft Azure web services.

C.2 Version 2 - Eclipse plugin and Weka

The difference between this version and the previous one is that the ML component has been changed, replacing Microsoft Azure Machine Learning with Weka. The ML models are created in the Weka Workbench and then used by the plugin since it contains the Weka libraries. This modification is because Weka is Opensource software, so we have more control over our algorithms and, therefore, over the models.

The metrics still have to be imported from a file in this version since it does not yet contain the metric extractor.

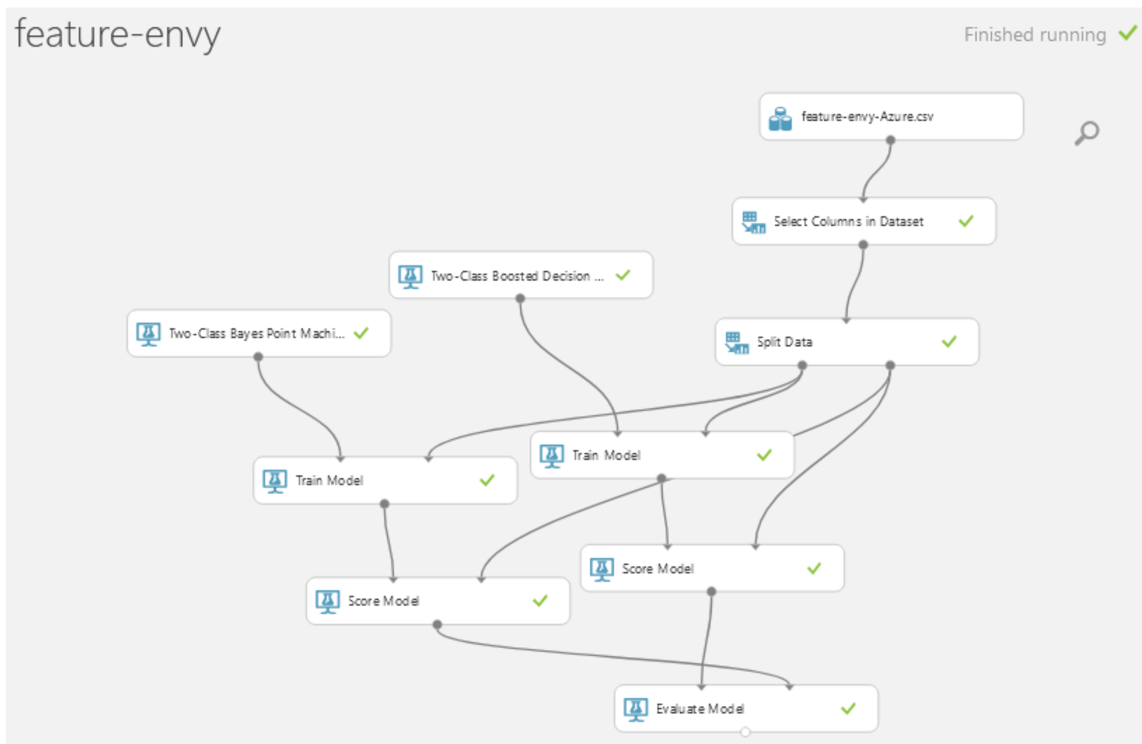


Figure C.2: Feature Envy training workflow

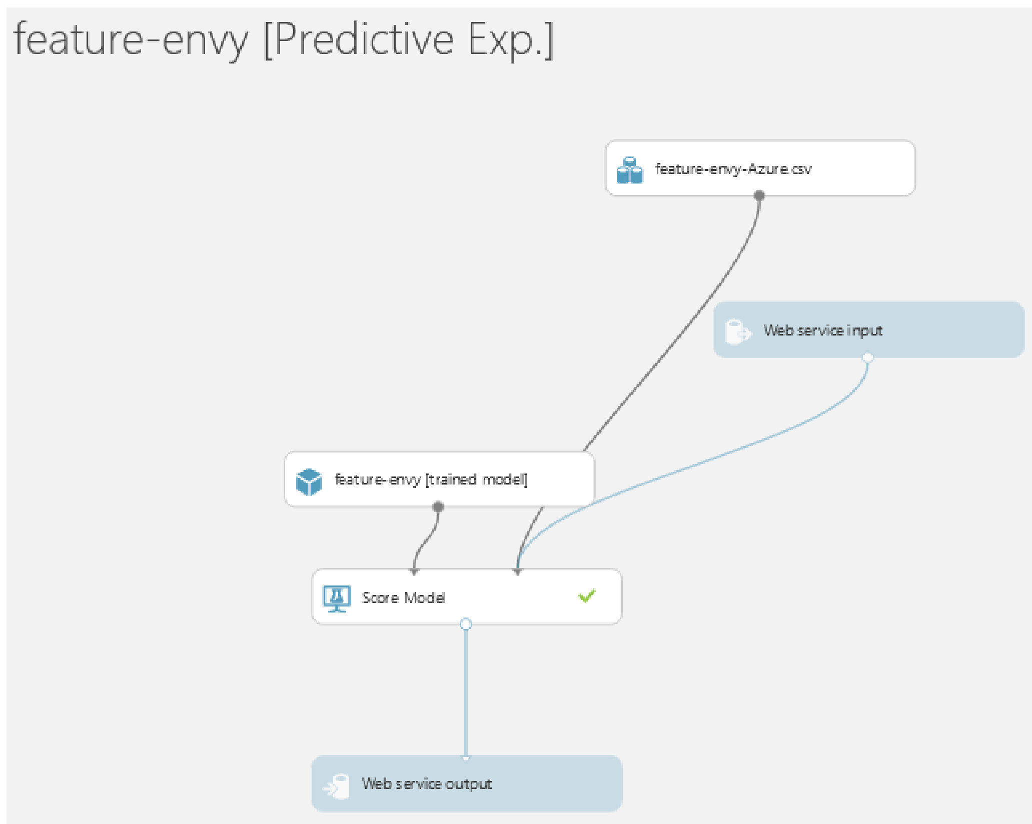


Figure C.3: Feature Envy predictive workflow

C.3 Version 3 - Microservices Architecture

This version is the one presented in chapter 5 and is in the development phase. However, all the components are already developed, missing the integration of these components, which is in the development phase.

The considerable advantage of this version, besides the microservices architecture, is the existence of the metrics extraction module, not being dependent on third-party applications. As explained in the 5.4.1 section, from the metamodel of the java Project being analyzed, a UML model is built, and metrics are extracted through OCL queries in the UML model. Thus, we can define in OCL any code metrics we need to detect any code smell by navigating the metamodel presented in Appendix D. Furthermore, this process aims to define more metrics as we increase the number of code smells the application detects. Right now, we are developing the application to detect 3 code smells (*God Class*, *Long Method*, *and Feature Envy*), but we plan to increase the number of code smells to be detected in the future and develop the respective metrics.

Appendix D.

Eclipse Java Metamodel

D.1 Eclipse Java Metamodel

This appendix introduces the Eclipse Java Metamodel. Figure D.1 represents the basic structure of a Java project and its hierarchical structure, including type inheritance and interface implementations. The contents of the Type metaclass, such as fields, methods, static initializers, and type parameters, are shown in figure D.2. The final diagram depicts the AST metaclasses (see Fig. D.3). More information about the Java Metamodel can be found in the study by Coimbra et al.[55].

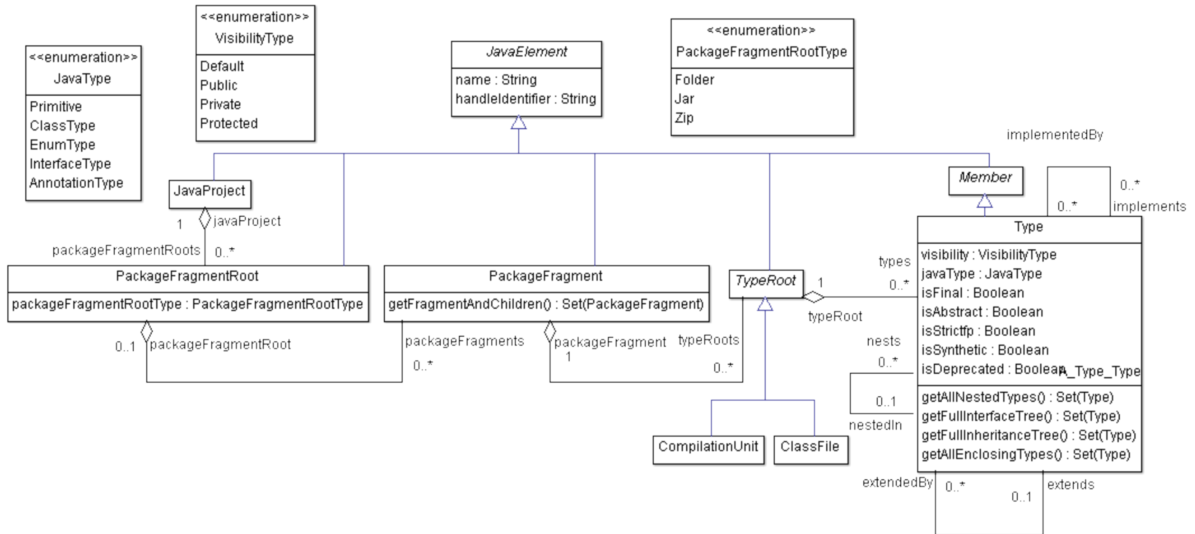


Figure D.1: Eclipse Java Metamodel - Java Project Structure

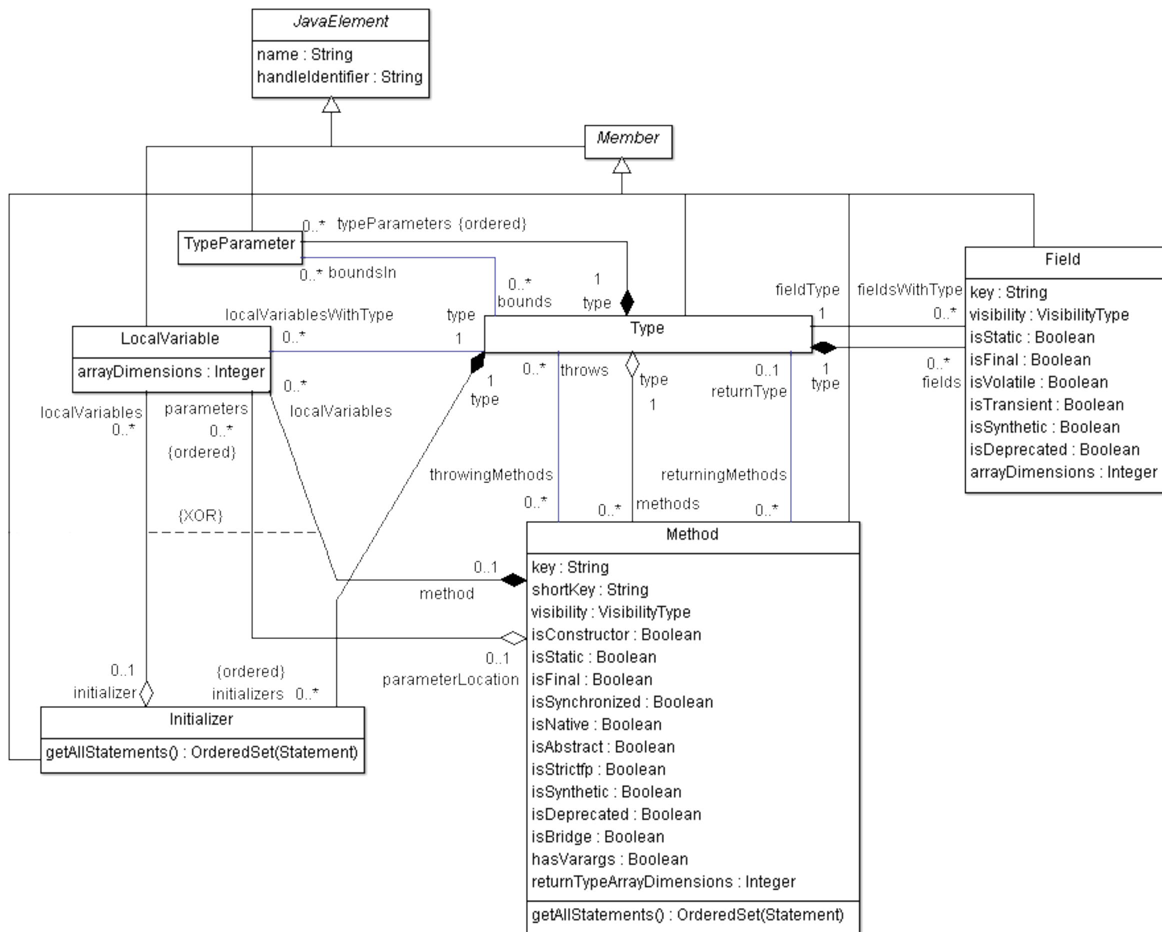


Figure D.2: Eclipse Java Metamodel - Type Components

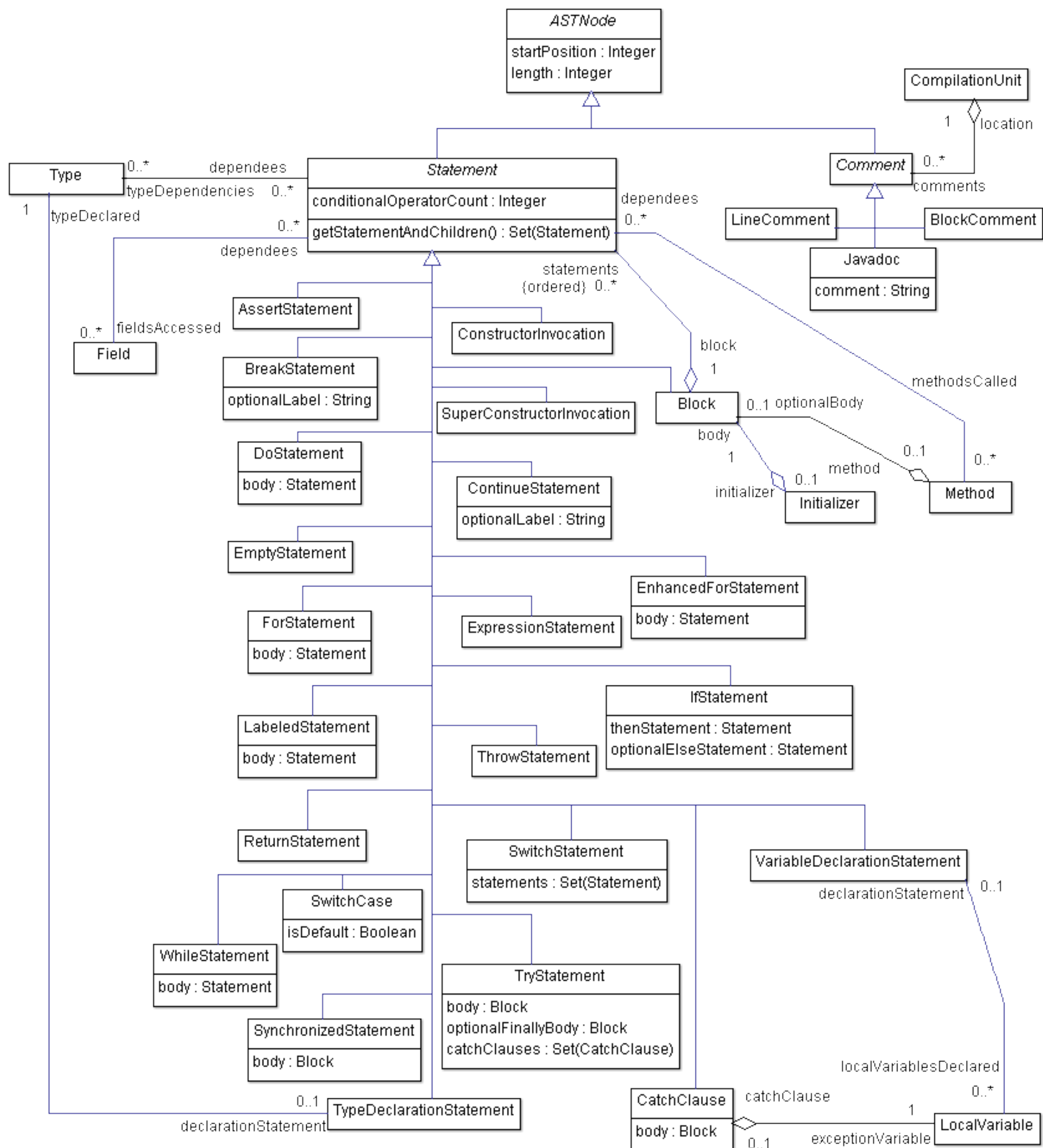


Figure D.3: Eclipse Java Metamodel - Abstract Syntax Tree Components

iscte INSTITUTO UNIVERSITÁRIO DE LISBOA

Streamlining Code Smells: Using Collective Intelligence and Visualization

José Pereira dos Reis