Programa de Pós-Graduação em

Computação Aplicada

Mestrado Acadêmico

Robson Keemps da Silva

SmellGuru: A machine learning-based approach to predict design problems

São Leopoldo, 2022

S586s Silva, Robson Keemps da
SmellGuru : a machine learning-based approach to predict design problems / by Robson Keemps da Silva. – 2022.
91 l. : ill., 30 cm.

Dissertação (mestrado) — Universidade do Vale do Rio dos Sinos, Programa de Pós-Graduação em Computação Aplicada, 2022.
Advisor: Prof. Dr. Kleinner Farias ; Co-advisor: Rafael Kunst.

1. Prediction. 2. Design problems. 3. Machine learning. 4. Code smells. 5. Bad smells. I. Title.

CDU 004

Cataloging-in-Publication:
Librarian Vanessa Borges Nunes - CRB 10/1556

ACKNOWLEDGEMENTS

Furthermore, this dissertation would not have happened without the great help of my advisor Kleinner Farias and co-advisor Rafael Kunst, who through long weekly lectures provided me with valuable help, advice and words that in one way or another motivated me to continue despite of all difficulties. I thank teacher Rodrigo Righi for his time and valuable suggestions. Finally, I would like to thank the entire collegiate of the program for their support during the course.

ABSTRACT

Nowadays, the prediction of source code design problems plays an essential role in the software development industry, identifying defective architectural modules in advance. For this reason, some studies have explored this subject in the last decade due to relation with aspects of maintenance and modularity. Unfortunately, the current literature lacks (1) a generic workflow approach that contains key steps to predict design problems, (2) a language to allow developers to specify design problems, and (3) a machine learning model to generate predictions of design problems. Therefore, this dissertation proposes ModelGuru, which is a machine learning-based approach to predict design problems. In particular, this study (1) introduces an intelligible workflow that provides clear guidance to users and facilitates the inclusion of new strategies or steps to improve predictions; (2) proposes a domain-specific language (DSL) to specify bad smells, along with a tool support; and (3) proposes a machine model to support the prediction of design problems. In addition, this study carried out a systematic review of the literature that allowed creating an overview of the current literature on the subject of predicting design problems. An exploratory study was carried out to understand the impact of the proposed DSL on three variables: correctness rate of the created specifications, error-rate and time invested to elaborate the specifications of design problems. The initial results obtained, supported by statistical tests, point to for encouraging results by revealing an above correct rate than 50%, error rate below 30% and effort less than 15 minutes to specify a bad smell. The evaluation of the proposed SmellGuru approach was carried out with 23 participants, students and professionals from Brazilian companies with professional experience in software development. It was possible to assess the perceived ease of use, perceived usefulness and behavioral intention of using the proposed SmellGuru approach. Respondents agree that SmellGuru is easy to interpret (43.47%), Innovative (60.86%) and would make the software easier to maintain (78.26%). Finally, this study draws up some implications and shows the potential of adopting the proposed approach for supporting the specification and prediction of design problems.

Keywords: Prediction. Design Problems. Machine learning. Code Smells. Bad Smells.

LIST OF FIGURES

1 The systematic mapping process used in our study (adapted from (PETERSEN et al., 2008)). . . . 26
2 The selected studies throughout the filtering process. . . . 29
3 Distribution of the primary studies based the explored research topics over the years. . . . 36
4 Bubble chart that shows the relationship among three variables. . . . 37
5 An overview of the proposed approach. . . . 44
6 A component-based architecture for the proposed SmellGuru approach . . . 48
7 Diagram of *railroad* of the syntax of the language. . . . 50
8 Code example of SmellDSL. . . . 51
9 SmellDSL tool integrated into the Eclipse platform. . . . 52
10 Visualization SmellGuru . . . 56
11 Building a Prediction Model SmellGuru . . . 58
12 Confusion matrix results of best RF model . . . 59
13 Feature importance to the RF algorithm . . . 60
14 Confusion Matrix - SVM with RBF Kernel . . . 61
15 Confusion Matrix - SVM with Linear Kernel . . . 61
16 Average time invested per scenario in minutes (RQ3) . . . 67
17 Diagram *SmellDSL* 01 . . . 91

LIST OF TABLES

1	Research questions that were investigated in this article	25
2	A description of the major terms and their alternative terms.	26
3	List of the used search engine.	27
4	List of the selected studies.	31
5	Classification of the primary studies based on their design problems (RQ1).	32
6	Classification of the primary studies based on their prediction aspects (RQ2).	33
7	Classification of the primary studies based on their prediction techniques (RQ3).	34
8	Study classification by contributions (RQ4).	35
9	Study classification by research methods (RQ5).	35
10	Place of publication of studies (RQ6).	37
11	Comparative analysis of related works	42
12	Description of Class level Design Metrics with categorization.	55
13	Metrics of the RF model with the highest accuracy	60
14	Accuracy of trained models	62
15	Evaluation scenarios of *SmellDSL*	65
16	Participants profile	66
17	Initial results for the tested hypotheses.	66
18	Participants profile	71
19	Participants Experience	72
20	TAM (Technology Acceptance)	73
21	TAM (Clarity)	73

CONTENTS

1 INTRODUCTION . . . **13**
1.1 Problem Statement . . . 14
1.2 Research Questions . . . 16
1.3 Objectives . . . 17
1.4 Methodology . . . 17
1.5 Outline . . . 18

2 BACKGROUND . . . **19**
2.1 Software design . . . 19
2.2 Design problems . . . 19
2.3 Prediction of design problems . . . 20

3 RELATED WORK . . . **23**
3.1 Mapping of Literature . . . 23
3.1.1 Planning . . . 25
3.1.2 Objective and research questions . . . 25
3.1.3 Search strategy . . . 26
3.1.4 Elaboration of the Search String . . . 26
3.1.5 Exclusion and inclusion criteria . . . 27
3.1.6 Data extraction . . . 28
3.1.7 Study Filtering . . . 29
3.1.8 Results . . . 30
3.1.9 RQ1: What are the design problems explored by prediction techniques? . . . 30
3.1.10 RQ2: What aspects are considered for predicting design problems? . . . 32
3.1.11 RQ3: Which techniques have been used to predict design problems? . . . 33
3.1.12 RQ4: What would be the contributions? . . . 34
3.1.13 RQ5: What research methods were used? . . . 34
3.1.14 RQ6: Where have the studies been published? . . . 35
3.1.15 Discussion and future directions . . . 37
3.1.16 Distribution of primary studies . . . 38
3.1.17 Future challenges . . . 38
3.1.18 Threats to validity . . . 39
3.2 Analysis of the Literature on Domain-Specific Languages for Specifying Bad Smells . . . 39
3.2.1 Analysis of related works . . . 40
3.2.2 Comparative analysis of the selected related works . . . 41

4 PROPOSED APPROACH . . . **43**
4.1 Overview of the SmellGuru approach . . . 43
4.1.1 Component-based architecture . . . 46
4.2 Domain-Specific Language for Specification of Bad Smells . . . 48
4.2.1 Language Design Decisions . . . 48
4.2.2 Language Grammar . . . 49
4.2.3 Implementation Aspects . . . 50
4.3 Machine Learning Model for Predicting Design Problems . . . 51
4.3.1 Methodology . . . 52
4.3.2 Classifier - Random Forest Algorithm . . . 53

4.3.3 Predictive model . . . 54
4.3.4 Description of Dataset . . . 54
4.3.5 Implementation Aspects . . . 55
4.3.6 A Proposal of SmellGuru Dashboard . . . 55
4.3.7 An extension of the approach . . . 56
4.3.8 Experimental Design . . . 57
4.3.9 Operation . . . 58

5 EVALUATION . . . 63
5.1 Evaluation SmellDSL . . . 63
5.1.1 Research Objective and Questions SmellDSL . . . 63
5.1.2 Study Variables . . . 63
5.1.3 Hypotheses and Analysis Procedure . . . 64
5.1.4 Experimental Tasks . . . 64
5.1.5 Context and Selection of Participants . . . 64
5.1.6 Results SmellDSL . . . 65
5.1.7 Conclusion and Future Works . . . 66
5.2 Evaluation of Model SmellGuru for Predicting Design Problems . . . 67
5.2.1 RQ1: Can the presence of code smells impact the design? . . . 67
5.2.2 RQ2: What is the performance of ML algorithms for predicting impact and non-impact design changes? . . . 68
5.2.3 RQ3: What features are the best indicators of change that impact design? . . . 68
5.2.4 Discussion . . . 68
5.3 Evaluation SmellGuru Proposed Model . . . 69
5.3.1 Context and Selection of Participants . . . 70

6 CONCLUSION AND FUTURE WORK . . . 75
6.1 Contribuitions . . . 76
6.2 Limitations . . . 76

REFERENCES . . . 79
.3 Grammar *SmellDSL* . . . 89
.4 Diagram *SmellDSL* . . . 90

1 INTRODUCTION

The **software development** process has evolved a lot since its inception. However, with the increasing presence of software in people's daily lives, especially in the execution of critical tasks (such as controlling aircraft and medical equipment), the concern with the final quality of the software has, since then, aroused the interest of industry and academia (SOMMERVILLE, 2011). It is often necessary to change the software design even at an advanced stage of development. This is because new functionalities are linked to the software, foreseen or not in its initial design stage, or correction of structural problems and even code maintenance. Coding can be distant from the initial project, its quality can be degraded and later its maintenance becomes difficult at the project level with several collaborators involved. In this context, it is important to investigate and propose new approaches to minimize coding and software design problems. Software developers need in their day-to-day to identify bad smells and software design problems, in modified and adapted code, increasing its complexity.

Design problems are internal structures of source code that challenge design principles or rules (OIZUMI et al., 2016; SHARMA; SPINELLIS, 2018; SURYANARAYANA; SAMARTHYAM; SHARMA, 2014). Design problems are often harmful in several software systems and having negative consequences, they are often targets of significant maintenance effort of software (SCHACH et al., 2002; GARCIA et al., 2009; SOUSA et al., 2018; TAKAHASHI et al., 2021). *Bad smells* are internal source code structures that challenge design principles or rules, negatively impacting internal quality (OIZUMI et al., 2016; SURYANARAYANA; SAMARTHYAM; SHARMA, 2014). Studies show that the presence of bad smells is related to aspects of (WERNER et al., 2020) maintenance, architectural integrity (FONTANA et al., 2017), in addition to susceptibility to failures (WALTER; ALKHAEIR, 2016). They often indicate potential problems in software, which may lead to long-term challenges and expensive maintenance efforts. Although bad smells often occur in source code, bad smells also exist in representations of design descriptions and models (POPOOLA; ZHAO; GRAY, 2021).

Nowadays in many projects, a good part of the cost is dedicated to maintenance. Thus, it is necessary to facilitate the maintainability and readability of the code, improving its internal quality (REIS; ABREU; CARNEIRO, 2017; SAHIN et al., 2014; LIU et al., 2011; DAS; YADAV; DHAL, 2019; KESSENTINI; VAUCHER; SAHRAOUI, 2010; GRIFFITH; WAHL; IZURIETA, 2011). According to (YAMASHITA; MOONEN, 2012a; KREIMER, 2005; DI NUCCI et al., 2018; MUMTAZ et al., 2018; SHIPPEY; BOWES; HALL, 2019; FONTANA; ZANONI, 2017; TOLLIN et al., 2017; IRWANTO, 2010; VIDAL et al., 2015), there are tools and techniques to assist professionals in the journey of identifying bad smells. According to (LIPOW, 1979; FENTON; NEIL, 1999; PECORELLI; Di Nucci, 2021; WANG; BANSAL; NAGAPPAN, 2021), predicting software design problems is still a complex task, with many challenges. The foregoing considerations show the importance of the concept of predicting software design problems; however, it is necessary to find out how a given design

problem can in the future generate code smells, it is possible to perform the prediction using machine learning techniques.

The literature covers a wide variety of ML techniques, each with characteristics and applications. Predictive models are generally data-driven models that require a variety of data streams data provided by multiple sources offline and in real time(DALZOCHIO et al., 2020).

Nowadays, the prediction of source code design problems plays an essential role in the software development industry, identifying defective architectural modules in advance. For this reason, some studies have explored this subject in the last decade due to relation with aspects of maintenance and modularity. Unfortunately, the current literature lacks (1) a generic workflow approach that contains key steps to predict design problems, (2) a language to allow developers to specify design problems, and (3) a machine learning model to generate predictions of design problems. Therefore, this dissertation proposes SmellGuru, which is a machine learning-based approach to predict design problems. In particular, this study (1) introduces an intelligible workflow that provides clear guidance to users and facilitates the inclusion of new strategies or steps to improve predictions; (2) proposes a domain-specific language (DSL) to specify bad smells, along with a tool support; and (3) proposes a machine model to support the prediction of design problems. In addition, this study carried out a systematic review of the literature that allowed creating an overview of the current literature on the subject of predicting design problems.

1.1 Problem Statement

Software development plays a crucial role in diverse fields of modern society. Solutions in electronics, transportation, healthcare, telecommunications, industry 4.0, and financial services rely on software for correct operation (SOMMERVILLE, 2011; HERMANN; PENTEK; OTTO, 2016; LEW et al., 2019; TSENG et al., 2021; SAAD; BAHADORI; JAFARNEJAD, 2021). Updating and maintaining these pieces of software is challenging and may lead to development-related problems. One way of identifying these problems is through the detection of code smells. The typical approach to find code smells involves software analytics and following good practices during the development process. An incipient field of research is the use of machine learning models to identify code smells. Related work in this field is still either theoretical or in the initial stages of development (FONTANA; BRAIONE; ZANONI, 2012; PAIVA et al., 2017; RASOOL; ARSHAD, 2017; AZEEM et al., 2019; KAUR et al., 2021a).

Usually, the source code of applications undergoes constant changes to accommodate new features or even correct existing ones. Such changes often promote the scattering and tangling of software concerns, which violate design principles such as the single-responsibility principle. These violations are typically noticed by bad smells (OIZUMI et al., 2016), symptoms of poor design (PALOMBA et al., 2017).

Empirical studies (PALOMBA et al., 2017, 2018; OIZUMI et al., 2016) point out that bad

smells are indicators of design problems. Couplers and bloaters are examples of such bad smells (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014) that indicate excessive couplings and large proportions of source code elements, turning maintenance tasks prone to errors. The documentation of architectural decisions, for example, through UML models (PETRE, 2014), might help developers refactoring tasks, however in some cases, such documentation is either not elaborated or out of date, leading to relying on the tacit knowledge of the developers to avoid new violations of design principles.

Prediction of design problems can play an essential role in this context, especially identifying defective architectural modules in advance. The greater the forecast of the appearance of design problems, the greater the anticipation capacity to address such problems. An alternative would be to generate predictions of design problems, for example, based on size, complexity, coupling, and cohesion as the source code is modified(DINAN et al., 2021).

The presence of design problem issues is an indication that the software architecture may be undergoing a process of degradation. Architectural degradation occurs when the implementation does not conform to the design decisions portrayed in the software architecture (TAYLOR, 2019). The longevity of an evolving application depends largely on its resilience to the symptoms of (MACIA et al., 2012a) architecture degradation. Therefore, it is important to have mechanisms that identify possible design problems that can generate **code smell**.

Different types of code smells are referenced in several works (EMDEN; MOONEN, 2002; FARD; MESBAH, 2013; YAMASHITA; MOONEN, 2012b; KHOMH; DI PENTA; GUEHENEUC, 2009; SJøBERG et al., 2013). While code smell detection and removal has been well researched over the past decade, it remains open to debate whether or not code smells should be considered significant conceptualizations of code quality issues based on software design.

However, reference techniques and architectures have been developed in the last decades, the works still have gaps to be filled by new investigations. The works found, for the most part, only classify or identify the smells (ABBES et al., 2011; BOUSSAA et al., 2013; AZEEM et al., 2019; Uchôa et al., 2020). Most recommend the use of static source code analysis tools and do not show a possible root of the actual problem detected. Furthermore, identifying design problems and demonstrating possible code anomalies that may be generated in the future is one of the issues to be addressed in this dissertation, so we can support the prediction of design problems and possibly identify the source of code smells. About the problems characterized above, this study investigates the Problems (P) that are described below:

- **P1: Lack of the current literature map on the automatic prediction of design problems.** Absence of a broad view of the state of the art. In recent years, several studies have analyzed various code smells, mainly using tools embedded with the use of static software code analysis. Little is known about the types of bad smells and the software metrics that characterize them, a clear definition of the smell, and the type of research strategies and software abstraction criteria that studies have used. Thus, it becomes relevant to explore the gaps and trends in the academy, as well as highlight the remaining opportunities.

- **P2: Lack of an overview about approaches to predict design problems.** The main difficulty is defined by the lack of an intelligible workflow to predict design problems, along with automatic strategies for identifying design problems.

- **P3: Lack of a machine learning-based approach to predict design problems.** Provide information from automatic techniques, as well as information obtained from real systems for the prediction of design problems according to the proposed definitions and particularities of each project.

- **P4: Lack of a domain-specific language for specifying bad smells.** Today, the literature still lacks approaches that help developers to more rigorously specify such bad smells. Furthermore, and lacks a technical component to measure understanding of a specific definition of bad smells, which is an important resource in the software design and maintenance process. Understanding and specifying a bad smell becomes an error-prone task.

1.2 Research Questions

This general section presents the research question that will be explored in the work, which seeks investigator how to propose a machine learning-based approach predict design problems?. To answer this research question, it is necessary to review the current literature related to predicting design problems, identify an intelligible workflow for predicting design problems, propose a machine learning model for predicting design problems, as well as propose a domain-specific language to specify bad smells.

> **General Research Question:** How to propose an approach based on machine learning to predict design problems?

After presenting the general research question, to explore different facets of this general research question 04 (four) specific questions were formulated to investigate the problems presented in Section 1.1.

- **RQ1:** How to propose a mapping of literature that has a new approach or techniques to identify design problems?

- **RQ2:** How to propose an approach that defines an intelligible workflow for predicting design problems?

- **RQ3:** How do build a machine learning model to predict design problems?

- **RQ4:** How to propose a domain-specific language for specifying bad smells?

1.3 Objectives

Section 1.1 presented the problem statement and emphasized the research questions, this section presents the objectives explored throughout this study. In addition, the main objective of this study is described below:

> **General Objective:** To support the prediction of software design problems, based on machine learning techniques

Thus, to answer the research questions, it is identified specific objectives (OS), derived from the main objective, and they are:

- **Obj1: Identify in the literature possible techniques for predicting design problems.** This objective aims to propose the use of machine learning techniques to evaluate the data catalog of the use case proposed in this work, this objective will be detailed in Chapter 3.
- **Obj2: Propose an intelligible workflow for predicting design problems.** This objective aims to present an overview of an approach that can present the main steps regarding the prediction of design problems. This objective will be detailed Chapter 4.
- **Obj3: Propose a catalog that defines design problems that can influence code smells.** This objective aims to make information obtained from real systems available for the prediction of design problems using machine learning techniques. This objective will be detailed in Chapter 4 and evaluation in Chapter 5.
- **Obj4: Propose a domain-specific language to define bad smells according to the needs of each software project and its possible variations.** This objective aims to propose a DSL to define bad smells already cataloged. This objective will be detailed in Chapter 4 and evaluation in Chapter 5.

1.4 Methodology

This section details the study methodology adopted to achieve the objectives described in Section 1.3. To explore the established objectives, the following research methods were defined:

The first phase of this study carried out a literature review on understanding design problems and code smells, techniques currently used to detect these problems based on work carried out between 2005 and 2020, using a systematic mapping review to carry out the first research question. This initial survey provided an overview of what was done, gathered information, and also introduced new research opportunities. In addition, this step was essential to get to know the area better.

The second phase, the creation of a predictive model that can be used to support the construction of the software project and identify possible code smells with the support of reverse

engineering and identification of possible design problems. Early identification using machine learning capabilities provides an enriched dataset that allows you to research the context behind design changes and potential impacts during the analysis process, with aspects of software processes. One of the main motivations for this early detection of impact changes in the project, thus being able to predict possible bad smells in the software project. After a thorough analysis, this step addressed the second and third research questions.

The third and final phase, a DSL was implemented to define bad smells. SmellDSL is a domain-specific language to assist developers in specifying bad smells. SmellDSL benefits developers by introducing notations to define bad smells and rules to identify them. The support tool, the SmellDSL tool, was implemented as an Eclipse Platform plugin. An exploratory empirical study was carried out with 12 participants, who used the SmellDSL tool to specify 8 bad smells, generating 96 evaluation scenarios, this step answered the fourth research question.

1.5 Outline

This work is organized into six chapters, as follows. After this first introductory chapter is **Chapter 2**, which concepts are the basic concepts for understanding the research; **Chapter 3** presents a comparative analysis of the state of the art in smell detection and prediction; **Chapter 4** discusses the implementation of the SmellGuru approach; **Chapter 5** addresses the evaluation of the proposed approach and, finally, **Chapter 6** presents a conclusion of this work, as well as contributions, suggestions and suggestions for future work of this research.

2 BACKGROUND

This chapter presents the main concepts of the concepts used in this work. A theoretical foundation of this research is organized into three sections. Section 2.1 defines the concepts of Software design. Section 2.2 discusses the main concepts of design problems. Section 2.3 presents the prediction of design problems and its main problems.

2.1 Software design

Software design is considered the most challenging task in software development. For example, it provides a bridge between satisfactions of system's critical requirements to implementation of software (BASS; CLEMENTS; KAZMAN, 2003). Over many years, based on their experience, software developers have encapsulated and suggested the proven solutions to satisfy the recurring design problems (HUSSAIN; KEUNG; KHAN, 2017).

The core of every software system is its architecture. Designing software architecture is a demanding task requiring much expertise and knowledge of different design alternatives, as well as the ability to grasp high level requirements and piece them together to make detailed architectural decisions. In short, designing software architecture takes verbally formed functional and quality requirements and turns them into some kind of formal model that is used as a base for code (RäIHä, 2010).

2.2 Design problems

A number of systematic literature reviews (SLRs) have been conducted in the area of **bad smells** (SOUSA; BIGONHA; FERREIRA, 2018; CARAM et al., 2019; MUMTAZ; SINGH; BLINCOE, 2021; ALJEDAANI et al., 2021). The **bad smell** detection process has motivated many researchers to propose different methods to deal with the occurrence of code smells in systems. Nowadays, machine learning techniques are utilized to address code smell issues with promising results. A machine learning classifiers needs first to be trained using a set of code smell examples to generate a model. The generated models are then used to identify or detect code smells in unseen or new instances (AL-SHAABY; ALJAMAAN; ALSHAYEB, 2020).

Beck and Fowler describe twenty-two bad smells and associate them with refactoring strategies to improve the design. Consequently, code smell analysis opens up the possibility for integrating both assessment and improvement in the software maintenance process. Nevertheless, to achieve accurate maintainability evaluations based on code smells, we need to better understand the "scope" of these indicators, i.e. know their capacity and limitations to reflect software aspects considered important for maintainability. In that way, complementary means can be used to address the factors that are not reflected by code smells. Overall, this will help to achieve more comprehensive and accurate evaluations of maintainability of software (YAMASHITA;

MOONEN, 2012a).

Different sets of **design problems** are detected with different scopes at the code level. The conceptual model part, describing the enumerated types, shows the scopes we considered (system, subsystem, package, class, method, operation). Problems are latent in code; detection usually occurs very late, and then, solutions are very complex. As a consequence, the software quality is negatively affected and technical debt increases, so redoing the software becomes the most realistic option. We believe, in a comparative perspective, that while refactoring has been extensively adopted by the software industry (ALKHARABSHEH et al., 2019).

2.3 Prediction of design problems

Machine learning techniques are applied and widely used in various contexts and fields, we can predict, sort, filter and group data (KAUR et al., 2021b; CHOUDRIE et al., 2021; MORENO-INDIAS et al., 2021; HAJI; AMEEN, 2021).

Currently, we live in a time where we are surrounded by a large volume of generated data for the advancement of collaborative development in software projects. The data produced by social platforms for collaborative development, have data with significant values that can help companies act preventively in relation to design problems and code smell, foreseeing certain situations. In this way, predictive analytics can be used, to extract information from a software project dataset in order to determine patterns and future results of the project.

The most popular learning approach in machine learning is supervised learning, in which the output is graded based on the input using a qualified data set and a learning algorithm. Classification and regression learning are two types of supervised learning. While in Unsupervised Learning, there are no output data for such input variables. Most data is unmarked, in which the machine attempts to detect the correlations between this data collection. It classifies them as clusters of various classes (STOIAN, 2020).

Predictive analytics can be understood as a process that allows you to discover the relationship between the examples of a dataset, described by a series of features (descriptive attributes), and the labels associated with them (class attributes). The clearest way to understand the prediction method is starting from two situations: if we are wanting to carry out a characterized prediction as a regression, from numerical data, or if we are wanting to perform a prediction of the type of classification, which starts from categorical data.

Predictive modeling is performed through a series of analytical and statistics, used to develop models that can predict future events from daily behaviors, including series analysis temporal or regression models. There are different forms of predictive models that vary according to the event or behavior being predicted. Almost all predictive models produce a score. a higher score indicates that a given event or behavior is very likely to occur. In order to bring assertive information, this approach uses large repository mining to support prediction of design problems and code smells. These statistical tools serve to build a classification model and regression

model based on information from a large mass of historical data, enabling the view of stronger scores for each modeling performed, in order to identify a model with predictive values.

The prediction goal is to predict which components on the next version of the software can have design problem. Given a particular software component, the classification problem of defect prediction is to determine what is the state of the said component (JIARPAKDEE et al., 2020; LUJAN et al., 2020; ZHANG et al., 2021; KOKOL; KOKOL; ZAGORANSKI, 2021).

The prediction of design problems is an attempt to anticipate design problems based on metrics using specific techniques. Some techniques analyze software project histories, trying guessing the behavior of the software in the future with respect to its coding. Predicting the location will increase the chances of identifying possible anomalies in the source code, in addition to assisting in testing activities, which aim to identify possible problems with the quality of the software. There are several benefits of prediction, such as support and planning for software testing, identification of code snippets where improvements are needed, reduction of code defects and mainly planning of future efforts within the software project (THOTA et al., 2020; KANG; RYU; BAIK, 2021; GIRAY, 2021).

The problems reported by (FENTON; NEIL, 1999), still persist to the present day. Even if we knew exactly the number of residual defects in our system we have to be extremely wary about making definitive statements about how the system will operate in practice. It is thus difficult to predict which defects are likely to lead to failures.

In more recent work, an attempt to improve defect prediction by taking smell information into account was presented by (PALOMBA et al., 2017). The authors found that prediction models that used smell information as an additional predictor variable had increased accuracy compared to other baseline models. A similar approach was proposed in (LIU et al., 2018) to predict change-prone files, where different smell-based metrics for effort-aware structural change-proneness prediction were examined.

3 RELATED WORK

In this chapter, a comparative analysis of the related works used in this research is carried out. This analysis aims to identify common criteria among related works raised from a study on the state of the art in the theme of predicting source code design problems. Therefore, this chapter is organized as follows: in the section 3.1 an analysis of the state of the art in the selected context of this work is presented; In this Section 3.2 describes related works identified in digital repositories, such as Google Scholar, Scopus (Elsevier) and arXiv, by applying the search string "DSL AND BAD SMELLS".

3.1 Mapping of Literature

In the scope of this work, a systematic mapping study was applied, a methodology proposed through the evidence-based paradigm that guides the analysis and development of a research topic (KITCHENHAM; BUDGEN; BRERETON, 2011), aiming to present an overview of a specific research area, identifying the type of research, its results and the number of publications available (GONÇALES et al., 2019).

Current studies pay close attention to empirical studies' elaboration to produce evidence-based knowledge and use purely statistical methods to predict the quality of the source code. However, little has been done to create a systematic map of studies published in recent decades. There are studies in the literature that also point to problems that impact the final quality of the software, (ALENEZI et al., 2016), (BOEHM; ROSENBERG; SIEGEL, 2019), (CHEN et al., 2018), (IBARRA; MUÑOZ, 2018), (MARTÍNEZ-FERNÁNDEZ et al., 2019), (WONG; YU; TOO, 2018), these studies mainly deal with the final quality of the software. The team involved in the software design must have version control of the artifacts and the software itself, a prerequisite for its modeling and construction to identify possible failures that may occur.

Besides, research is being developed in the area using **software analytics** (ABBES et al., 2011) and (BOUSSAA et al., 2013), tooling supports, and good development practices with improved source code quality. Barbosa *et al.* (BARBOSA et al., 2020) report that software design quality degradation can be avoided, reduced, or accelerated depending on the developers' communication dynamics and on specific roles performed within the software project. Understanding the role of communication dynamics and the content involved in the discussion is important to avoid software design anomalies. Certain social metrics can also be indicators of design decay when analyzing the two aspects together. Thus, it is important to define new approaches to spot them whenever possible. A number of systematic literature reviews (SLRs) have been conducted in the area of bad smells (PAULO SOBRINHO; DE LUCIA; ALMEIDA MAIA, 2018; FERNANDES et al., 2010, 2016; RASOOL; ARSHAD, 2015; LACERDA et al., 2020; SANTOS et al., 2018; SABIR et al., 2019).

Unfortunately, the authors focus on two categories of symptoms: low level and high level

structural bad smells. These symptoms occur in large and complex classes possibly due to the accumulation of responsibilities. These symptoms of such categories can be detected automatically using code analysis tools. Since the code review also aims to improve the quality design It is possible to observe many social aspects around a discussion such as: different roles of participants (central development of the organization and employees and users); temporal aspects (the time interval of the pull request); and the size and content of comments (snippets being used). It is suggested that even in situations where the number of comments is low but the number of words per comment is high there is a large volume of information that can be related to complexity of the change in the software. In general the content of the discussion can indicate the increase (number of words under discussion) and decrease (number of words per comment under discussion)possible decadence of software design.

It is appointed in (Uchôa et al., 2020) that existing studies tend to analyze the degradation of the software design and considering only unique events like introducing a single design problem or simply analyzing the degradation frequent. However understanding how design degradation evolves over time between reviews and in reviews of major importance. The impact of modern code review on the evolution of project degradation is analyzed. Since the code review also aims to improve the quality design and evaluations can be expected to gradually reduce over time various symptoms of software degradation. To this end, they were investigated retrospectively 14.971 code reviews of seven software systems belonging to two large open source communities.

To overcome these limitations (AZEEM et al., 2019) the adoption of a broader set of metrics that explore different types of information as textual and historical and dynamic and separation of interests aspects can be beneficial in devising more effective solutions that are also closer to the way developers perceive and identify code problems .The need for targeted methods so that the developer be guided to make predictions about code issues that are relevant. The lack of research is highlighted as the approaches based on machine learning that can be adapted for prioritization purposes and notes on design problems. So a closer look at how machine learning techniques is discussed and your settings are necessary to correctly interpret your results. There is still a lack of understanding of the role of cluster technology in predicting code anomalies.

Providing an overview and discussing the use of machine learning with approaches in the field of bad smells work (AZEEM et al., 2019) presents a Systematic Literature Review (SLR) on Machine Learning Techniques for Smell Detection Code. It is considered articles published between 2000 and 2017. Starting from an initial set of 2.456 articles only 15 of them actually took machine learning approaches. These studies address four different perspectives: (i) code smells considered, (ii) configuring machine learning approaches, (iii) design of the evaluation strategies, and (iv) a meta-analysis of the performance achieved by the models hitherto proposed.

The analyzes carried out show that God Class and Long Method and Functional Decomposition and Spaghetti Code have been widely considered in the literature. Decision trees and

support vector machines are more machine learning algorithms commonly used to code smell detection. Models based on a large set variables had a good performance. JRip and Random Forest are the classifiers more effective in terms of performance. The analyzes also reveal the existence of several open questions and challenges that the research community should focus on the future. Providing an overview and discussing the use of machine learning with approaches in the field of design problems.

3.1.1 Planning

Figure 1 introduces the planning followed to run our study. This protocol addresses the steps and guidelines for conducting systematic mapping studies in software engineering (KEELE et al., 2007; KITCHENHAM; BUDGEN; BRERETON, 2011). Moreover, the defined protocol was inspired by previously published studies (GONÇALES et al., 2015; MENZEN; FARIAS; BISCHOFF, 2021; GONÇALES; FARIAS; SILVA, 2021).

3.1.2 Objective and research questions

This work aims to provide an overview of the current literature by filtering (Section 3.1.7) and classifying the articles currently available (Section 3.1.8), identifying potential gaps, challenges, opportunities, and promising directions for further research (Section 3.1.15). To address this objective, six research questions were formulated and motivated (Table 1). Answering these questions, this study seeks to investigate six research questions, including the types of design problems most commonly investigated, the prediction aspects considered to predict design problems, the prediction techniques used to anticipate the future occurrence of design problems, the main contribution reported in each selected study, the research methods applied to run the research and the research venue chosen to publish the articles.

Table 1: Research questions that were investigated in this article

Research Questions	Motivation	Variable
RQ1: What are the design problems explored by prediction techniques?	Reveal the most common types of design problems that have been most explored.	Types of design problems
RQ2: What aspects are considered for predicting design problems?	Understand the different aspects considered for predicting software design problems.	Prediction aspect
RQ3: Which techniques have been used to predict design problems?	Reveal the most commonly used comparison techniques.	Prediction technique
RQ4: What is the main contribution of the primary study?	Identify the main contributions of the current literature.	Main contribution
RQ5: What are the empirical methods used to evaluate the prediction of design problems?	Identify the research methods used to evaluate the prediction	Research method
RQ6: Where have the studies been published?	Reveal the target venues used to report the results.	Research venue

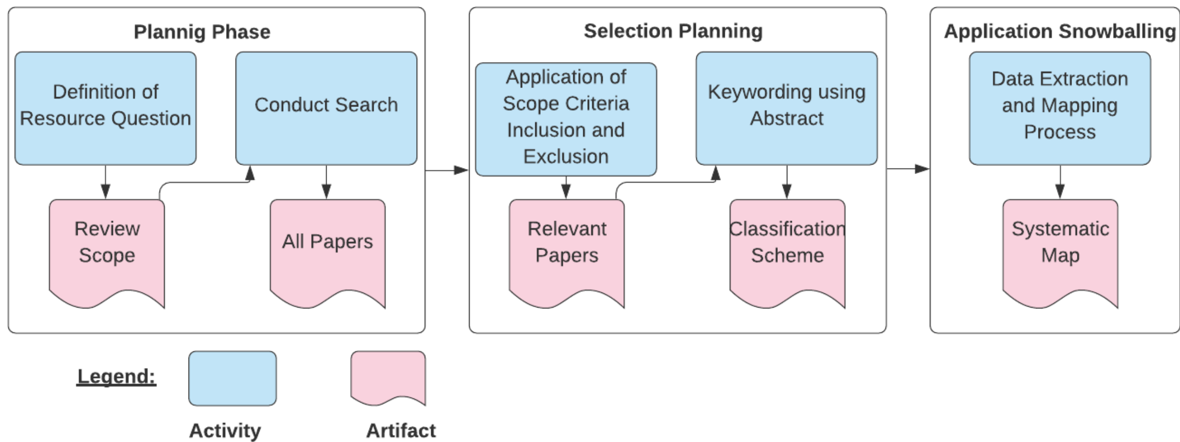

Figure 1: The systematic mapping process used in our study (adapted from (PETERSEN et al., 2008)).

3.1.3 Search strategy

After defining the research questions, the next step is to determine the set of main terms in the research questions to search for potentially relevant articles. The key terms were defined base on well-known empirical guidelines (WOHLIN et al., 2012; KITCHENHAM, 2012; PETERSEN; VAKKALANKA; KUZNIARZ, 2015; AL-QUDAH; MERIDJI; AL-SARAYREH, 2015). Table 2 shows the main terms and their alternative ones. The selection of these terms considered their relevance to the research field.

Table 2: A description of the major terms and their alternative terms.

Main Term	Alternative Term
Design problem	bad smell, code smell, code anomaly
Predict	forecast, foresee, anticipate
Maintenance	Evolution, Development, Review, Refactoring

3.1.4 Elaboration of the Search String

Steps to define search strings. The main steps followed to define the search string were: (1) Identify candidate keywords, reading studies (e.g., (SHARMA; SPINELLIS, 2018; PALOMBA et al., 2017; OIZUMI et al., 2016; FOWLER et al., 1999; SURYANARAYANA; SAMARTHYAM;

SHARMA, 2014)) chosen by relevance and convenience to pinpoint the main terms; (2) Identify closely related words and alternative terms or synonyms related to the candidate keywords; (3) Check through an initial search if the terms are in articles widely known in the field an interactive and incremental process run by the authors; and (4) Join the alternative terms using logical operator "**OR**", and the the main terms using logical operators "**AND**". Previous works (WOHLIN et al., 2012; PETERSEN; VAKKALANKA; KUZNIARZ, 2015) were also considered to formulate the search string. The combinations that produced the most significant results are shown as follows:
*(design problem OR bad smell **OR** code smell **OR** code anomaly) **AND** (prediction OR forecast **OR** foresee **OR** anticipate) **AND** (maintenance **OR** evolution **OR** development **OR** review **OR** refactoring)*

Electronic databases. After determining our search string, the next step was to identify the electronic databases and retrieve potentially relevant studies. Table 3 details the electronic databases used to search for studies for the preparation of systematic mapping. These electronic databases were selected for three reasons. First, these databases have a large and representative number of articles published related to the research topic explored in our mapping. Second, they have been used extensively in systematic mapping studies, pointing out their usefulness and effectiveness. Third, previous studies (EL KOUTBI; IDRI; ABRAN, 2016), (KITCHENHAM, 2012), (KITCHENHAM; BUDGEN; BRERETON, 2011), (PETERSEN; VAKKALANKA; KUZNIARZ, 2015), (KUUTILA et al., 2020), have demonstrated the effectiveness of such electronic databases used to perform literature reviews.

Table 3: List of the used search engine.

Source	Electronic Address
ACM DL	dl.acm.org
IEEE	ieeexplore.ieee.org
Science Direct	www.sciencedirect.com
Scopus	www.scopus.com
Elsevier	www.elsevier.com
Google Scholar	scholar.google.com

3.1.5 Exclusion and inclusion criteria

These criteria prescribe rules to make the process filtering as objective and auditable as possible, avoiding bias generally found in manual tasks performed by humans. For this systematic mapping, the inclusion criteria were applied directly to the electronic databases. For filtering studies, the inclusion (IC) and exclusion (EC) criteria are defined as obtaining significant results to filter studies that are not relevant to obtain answers to research questions. The inclusion criteria (IC) considered were:

- **IC1:** Published articles, journals, in an event or periodical that deals with the evaluation

of prediction techniques, or source code design problems, whether general-purpose or not;

- **IC2:** Studies published between January 2005 and March 2021;
- **IC3:** Studies published in Portuguese or English;
- **IC4:** Studies that contain key terms: Prediction, Bad Smell, Software, or Design.

The exclusion criteria (EC) considered were:

- **EC1:** The title, summary or even its content without relation to the search string;
- **EC2:** Short studies (up to 4 pages) written in another language, other than Portuguese or English;
- **EC3:** Duplicate studies;
- **EC4:** Abstract did not address any aspect of the research questions;
- **EC5:** Older versions of published studies prior to 2005;
- **EC6:** Studies that are narrowly related to Software Engineering, Software Development and/or contrary to research questions;
- **EC7:** The full text did not address issues considering the prediction of design models;

3.1.6 Data extraction

The data extraction procedures consist of a careful reading of each selected work and storing the extracted data in an on-line Google Spreadsheet. This spreadsheet served as a basis for the collection and synchronization of data extraction actions by the authors. Each primary study was carefully read and its data extracted to answer the research questions formulated. The extraction process was iterative and incremental, aiming that the authors could collect and audit the data. This made it possible to align the way data was collected and to detect any incorrect collection procedures.

In particular, the articles were classified according to the type of study performed (PETERSEN; VAKKALANKA; KUZNIARZ, 2015): (1) *Evaluation study:* a specific problem is defined, proposing a solution and conducting an empirical analysis, to point out the advantages and disadvantages; (2) *Philosophical studies:* a taxonomy or conceptual framework is proposed as a way to outline a research area; (3) *Experience article:* An experience report on the theme of prediction of design problems. Typically, these studies explain what and how something was done in practice; (4) *Opinion article:* Someone's personal opinion about predicting design problems. The report does not have a clear methodology, nor related work, focusing on the opinion itself; (5) *Solution proposal:* A proposed solution for a given problem is presented. The evaluation sticks to the execution of examples or the elaboration of prototypes, rarely to the

execution of robust empirical studies.; and (6) *Validation search:* Studies that typically perform experimental studies to evaluate solutions, approaches, techniques or processes that have not yet been used in real-world settings.

3.1.7 Study Filtering

The filtering process was made up of five steps performed sequentially. The focus was on selecting a sample of representative studies from a sample of potentially relevant ones. Figure 2 illustrates the results collected from the execution of each step. Each step is described as follows:

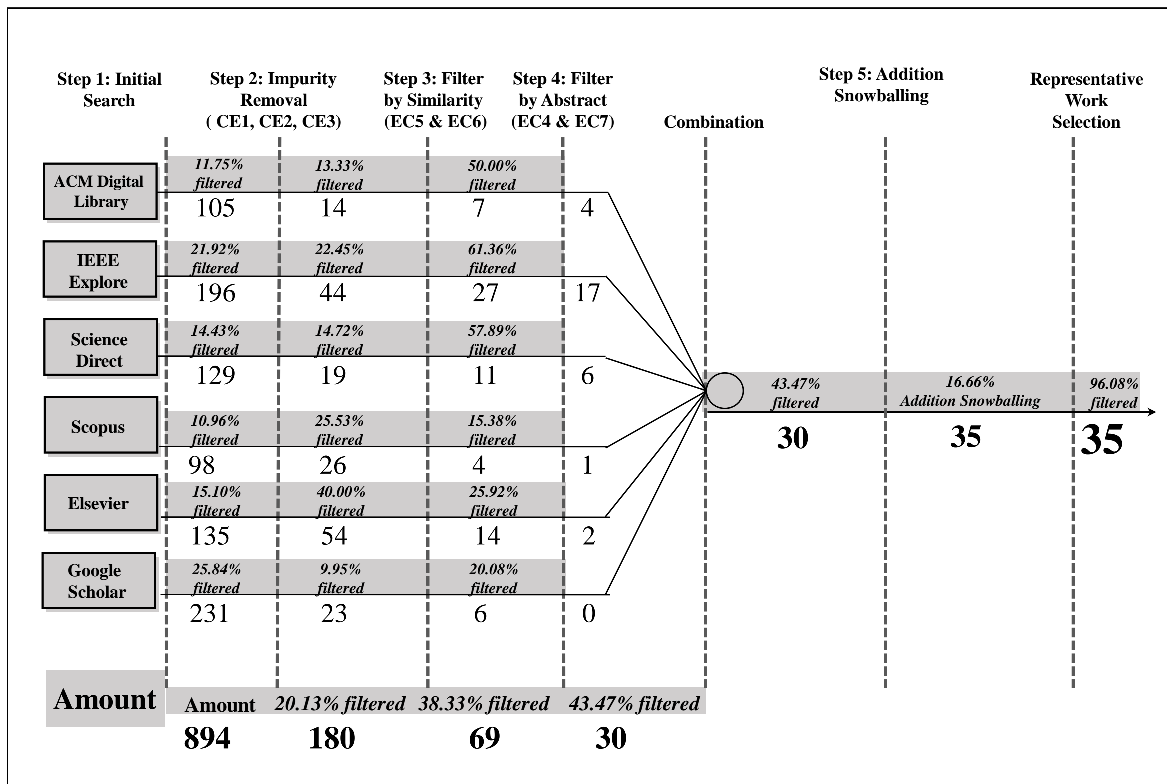

Figure 2: The selected studies throughout the filtering process.

- **Step 1: Initial search.** It gathers the initial results obtained after applying the search string in the electronic databases (Table 3). In total, 894 candidate studies were recovered.

- **Step 2: Exclusion criteria.** Three exclusion criteria (EC1, EC2, and EC3) were applied to remove impurities. Some studies were withdrawn due to the absence of any semantic relationship to its title, abstract, or even content, considering the theme investigated in this research (that is, out of scope). In addition, studies that were not written in English or Portuguese were also discarded. In total, 180 studies (20.13%) continued in the

next stage, while 714 works were discarded. Calls for conference articles, special issues of journals, patent specifications, research reports, and no peer-reviewed material were examples of discarded materials.

- **Step 3: Filter by similarity.** This step also discarded the studies that were selected by the search string, however, their content was not closely related to the research questions, or they had no close relationship with the study area, e.g., software development and prediction of design problems. The EC5 and EC6 were applied. For that, 38.33% (69 out of 180) of the studies were filtered.

- **Step 4: Filter by abstract.** Exclusion criteria (EC4 and EC7) were applied to remove the studies considering their abstract, and after their full text. In total, 39 studies were removed, leaving 30 studies (43.47%) for the next step.

- **Step 5: Addition by snowballing.** Some studies may not have been located, although the search engines used are widely qualified. To mitigate this threat, studies have been added using the snowballing method (both backward and forward) (WOHLIN, 2014; JALALI; WOHLIN, 2012). After selecting the studies in step 04, a manual analysis of the references and citations of the hitherto filtered studies was performed. Five studies were incorporated.

Finally, 35 studies were filtered as the most representative, hereinafter called *primary studies* (Table 4).

3.1.8 Results

This section presents the results obtained after classifying the primary studies (Table 4) to answer the formulated research questions (Table 1).

3.1.9 RQ1: What are the design problems explored by prediction techniques?

Table 5 presents the design problems investigated by the primary studies. The main feature is that the majority of the primary studies explored Bloaters (62.86%, 22/35), Architectural Problems (54.29%, 19/35), and Couplers (42.86%, 15/35). Note that the primary studies usually explored more than one design problem. In total, the problems were explored 79 times.

There are two interesting findings when comparing this result with studies already published. First, there may be a relationship between the most frequently explored design problems with the diffuseness of design smells. Previous empirical studies (PALOMBA et al., 2018; SJØBERG et al., 2012) revealed a relationship between diffusion of bad smells and the size and complexity of the source code. Palomba et al. (PALOMBA et al., 2018) point out that the smelly

Table 4: List of the selected studies.

ID	Title	Year	#Citations	#References
A1	Identifying Architectural Problems through Prioritization of Code Smells (VIDAL et al., 2016a)	2016	17	25
A2	Are SonarQube Rules Inducing Bugs? (LENARDUZZI et al., 2020)	2020	02	32
A3	Do Code Smells Impact the Effort of Different Maintenance Programming Activities? (SOH et al., 2016)	2016	28	40
A4	Code smells detection 2.0: Crowdsmelling and visualization (REIS; ABREU; CARNEIRO, 2017)	2017	3	50
A5	Code-Smell Detection as a Bilevel Problem (SAHIN et al., 2014)	2014	56	73
A6	LDFR: Learning deep feature representation for software defect prediction (XU et al., 2019)	2019	0	118
A7	Static Code Analysis of IEC 61131-3 Programs: Comprehensive Tool Support and Experiences from Large-Scale Industrial Application (PRÄHOFER et al., 2016)	2017	20	25
A8	BDTEX: A GQM-based Bayesian approach for the detection of antipatterns (KHOMH et al., 2011)	2011	106	33
A9	Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort (LIU et al., 2011)	2012	106	54
A10	Improving Design Smell Detection for Adoption in Industry (ALKHARABSHEH et al., 2018)	2018	02	28
A11	Detecting Code Smells using Deep Learning (DAS; YADAV; DHAL, 2019)	2019	0	27
A12	Deviance from perfection is a better criterion than closeness to evil when identifying risky code (KESSENTINI; VAUCHER; SAHRAOUI, 2010)	2010	73	28
A13	Evolution of legacy system comprehensibility through automated refactoring (GRIFFITH; WAHL; IZURIETA, 2011)	2011	13	30
A14	Automatically classifying source code using tree-based approaches (PHAN et al., 2018)	2016	8	42
A15	Detecting Android Smells Using Multi-Objective Genetic Programming (KESSENTINI; OUNI, 2017)	2017	13	36
A16	A hierarchical method for detecting codeclone (DEVI; PUNITHAVALLI, 2011)	2011	1	20
A17	Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems (MACIA et al., 2012b)	2012	93	49
A18	On the Relation between External Software Quality and Static Code Analysis (PLOSCH et al., 2008)	2008	14	19
A19	Do code smells reflect important maintainability aspects? (YAMASHITA; MOONEN, 2012a)	2012	165	40
A20	Adaptive Detection of Design Flaws (KREIMER, 2005)	2005	50	47
A21	Detecting code smells using machine learning techniques: Are we there yet? (DI NUCCI et al., 2018)	2018	43	90
A22	An empirical study to improve software security through the application of code refactoring (MUMTAZ et al., 2018)	2018	12	86
A23	Automatically identifying code features for software defect prediction: Using AST N-grams (SHIPPEY; BOWES; HALL, 2019)	2019	8	81
A24	Code smell severity classification using machine learning techniques (FONTANA; ZANONI, 2017)	2017	34	42
A25	Change Prediction through Coding Rules Violations (TOLLIN et al., 2017)	2017	6	12
A26	Visual Indicator Component Software to Show Component Design Quality and Characteristic (IRWANTO, 2010)	2010	2	11
A27	Iterative software fault prediction with a hybrid approach (ERTURK; SEZER, 2016)	2016	28	64
A28	Using (Bio)Metrics to Predict Code Quality Online (MULLER; FRITZ, 2016)	2016	32	77
A29	Less is more: Minimizing code reorganization using XTREE (KRISHNA; MENZIES; LAYMAN, 2017)	2017	15	68
A30	Bad-smell prediction from software design model using machine learning techniques (MANEERAT; MUENCHAISRI, 2011)	2011	40	14
A31	On the criteria for prioritizing code anomalies to identify architectural problems (VIDAL et al., 2016b)	2016	8	9
A32	Software Defect Prediction via Convolutional Neural Network (LI et al., 2017)	2017	84	50
A33	A Hybrid Approach To Detect Code Smells using Deep Learning (HADJ-KACEM; BOUASSIDA, 2018)	2018	5	37
A34	Predicting Design Impactful Changes in Modern Code Review: A Large-Scale Empirical Study (UCHÔA et al., 2021)	2021	0	76
A35	JSpIRIT: A Flexible Tool for the Analysis of Code (VIDAL et al., 2015)	2015	47	20

diffuseness is associated with the size and complexity of the source code. This smelly diffuseness typically addresses bloater smells, including Long Method, Large Class, Primitive Obsession, Long Parameter List, among others. Typically, these smells appear gradually throughout the source code, as the source code undergoes frequent evolution or maintenance tasks, remaining in the absence of a refactoring effort to eradicate them. Sjoberg et al. (SJØBERG et al., 2012) reveal that the size of classes often impacts maintainability more than the presence of bad smells. The result highlights a higher frequency of studies concerned with predicting the appearance of Bloaters. This concern makes sense when empirical findings have already revealed their harmful effect on maintainability.

The second finding would be that unlike exploring specific design problems, the primary studies explored more than one. On average, the primary studies investigated at least two design problems. Previous empirical findings already point out that the presence of multi-

ple code smells in classes tends to increase the change- and fault-proneness (KHOMH et al., 2012; PALOMBA et al., 2018), and design problems can arise from this clustering of code smells (OIZUMI et al., 2016). In this sense, exploring more than one design problem makes sense and would be supported by the findings already reported in the literature.

Table 5: Classification of the primary studies based on their design problems (RQ1).

Classification	Amount	Percentage	List of primary studies
Bloaters	22/35	62.86%	[A4], [A5], [A8], [A9], [A11-15], [A17-18], [A20-22], [A24-25], [A28-30], [A32-33], [A35]
Architectural Problems	19/35	54.29%	[A1], [A3-7], [A9], [A11-20], [A31], [A33], [A20-22], [A24-25], [A28-30], [A32-33], [A35]
Couplers	15/35	42.86%	[A3], [A5-6], [A9], [A15], [A17-18], [A20-22], [A29-30], [A32-33], [A34]
Dispensables	12/35	34.29%	[A5], [A16], [A19], [A21-23], [A29-30], [A32-33], [A34-35]
Object-Orientation Abusers	6/35	17.14%	[A1], [A23], [A26-27], [A30], [A35]
Change Preventers	4/35	11.43%	[A5], [A19], [A22], [A29]
Technical Debt	1/35	2.86%	[A2]

3.1.10 RQ2: What aspects are considered for predicting design problems?

Table 6 presents the commonly used aspects for predicting design problems in the primary studies. Understanding the considered aspects of the source code is essential to pinpoint which features are relevant, for example, to anticipate design problems.

The collected results indicate to a tendency to use structural property that can be calculated by metrics, including Code complexity (77.14%, 27/35), Size (77.14%, 27/35), Inheritance (60%, 21/35), Coupling (51.43%, 18/35), Cohesion (40%, 14/35), and Agglomerations (8.57%, 3/35) in a smaller amount. These results corroborate with previous studies, revealing that such structural properties can be predictors of design problems and bugs (PALOMBA et al., 2017; MOHA et al., 2009; ZIMMERMANN; PREMRAJ; ZELLER, 2007; NAGAPPAN; BALL; ZELLER, 2006). Palomba et al. (PALOMBA et al., 2017) use structural properties of source code to propose a smell-aware bug prediction model, including code complexity, coupling, cohesion, lines of code, coupling dispersion, among others. Zimmerman et al. (ZIMMERMANN; PREMRAJ; ZELLER, 2007) indicate a positive correlation between code complexity and bugs. Nagappan et al. (NAGAPPAN; BALL; ZELLER, 2006) also examined the use of metrics to predict buggy components across 5 Microsoft projects.

Moreover, the primary studies explored source code design problems computed from pure object-oriented code, e.g., pure Java code. However, real-world applications rarely have pure object-oriented code. Typically, software systems are built from the composition of pure object-oriented code together with numerous annotations of frameworks and architectural styles, such as @RestController and @PostMapping from Spring Platform — i.e., annotations for REST web controller and mapping HTTP requests onto specific handler methods, respectively. Thus, computing and predicting design problems from semantically enriched code would require un-

derstanding the meaning of annotations. For example, predicting design problems in source code with Spring Boot platform annotations would require dealing with semantic and structural aspects — a challenging and ever-present problem, since semantic information related to annotations is rarely formally specified.

Table 6: Classification of the primary studies based on their prediction aspects (RQ2).

Classification	Amount	Percentage	List of primary studies
Code complexity	27/35	77.14%	[A2], [A5], [A7-9], [A11-15], [A17], [A19-30], [A32-33], [A34-35]
Size	27/35	77.14%	[A2], [A5], [A7-9], [A12-25], [A27-30], [A32-33], [A34-35]
Inheritance	21/35	60%	[A5], [A7-9], [A11-13], [A15], [A17-18] [A21-24], [A26-27], [A29-30], [A32-33], [A35]
Coupling	18/35	51.43%	[A1], [A3], [A5], [A15-18], [A20-24] [A26-27], [A29-30], [A32], [A35]
Cohesion	14/35	40%	[A11-13], [A15], [A18-22], [A24] [A27], [A29], [A32], [A35]
Agglomerations	3/35	8.57%	[A1], [A31]
Others	11/35	31.43%	[A4-8], [A10], [A12-14], [A28], [A34]

3.1.11 RQ3: Which techniques have been used to predict design problems?

Table 7 introduces the collected data related to the techniques used to predict bad smells problems investigated by the selected studies. The main feature is that most primary studies use Machine Learning techniques (54.29%, 19/35), Decision Tree (20%, 7/35) and Random Forest (20%, 7/35), followed by Rules/Heuristics (17.14%, 6/35) and Prioritization Criteria (17.14%, 6/35) and Linear Regression (11.43%, 4/35) and Logistic Regression (8.57%, 3/35) and Bagging (5.71%, 2/35) and Others (20%, 7/35). Note that primary studies generally explored Machine Learning and used algorithms for training problem prediction models. It is extremely important to highlight and compare this result with the published study that notes that further studies are needed to consider the use of cluster learning, multiclassing and resource selection technique for code smells detection (AL-SHAABY; ALJAMAAN; ALSHAYEB, 2020).

In an attempt to anticipate the location of defects in an application through the use of specific techniques, the primary studies explored more than one bad smells according to the classification of Table 6, evaluating the results present in Table 7, we can say that apprenticeship is proposed to improve the performance of software problem classifiers, combining different classifiers and methods in defect prediction.

The results indicate that the use of Learning Techniques is part of an in-depth analysis of the performance index of software bug prediction models. Therefore, future efforts will be dedicated to analyzing the contribution of information related to the detection of bad smell in the context of models. Prediction of local learning bug. Finally, the future research agenda includes the definition of new factors that influence the performance of forecasting models (PALOMBA et al., 2017).

The vast majority of selected primary studies use the practice prioritizing bad smell, that is,

Table 7: Classification of the primary studies based on their prediction techniques (RQ3).

Classification	Amount	Percentage	List of primary studies
Machine Learning	19/35	54.29%	[A2-4], [A6], [A8], [A11-12], [A14], [A20-21], [A23-25], [A27-28], [A30], [A32-33], [A34]
Decision tree	7/35	20%	[A2], [A6], [A14], [A23-25], [A34]
Random forest	7/35	20%	[A2], [A6], [A21], [A24-25], [A30], [A35]
Rules/Heuristics	6/35	17.14%	[A5], [A8], [A11-12], [A14-15]
Prioritization Criteria	6/35	17.14%	[A1-2], [A6], [A12], [A24], [A31]
Linear regression	4/35	11.43%	[A2], [A24], [A11]
Logistic regression	3/35	8.57%	[A2-3], [A24], [A30]
Bagging	2/35	5.71%	[A2], [A8]
Others	7/35	20%	[A1], [A5-7], [A10], [A34]

prioritizing the groups of anomalies in the code of according to his criticism of the system's architecture, that is, its ability to point out architectural problems. Several automated approaches are proposed to generate rules that can detect bad smells in static software codes. A rule is a combination of quality metrics and their threshold values to detect a specific type of Bad Smells. The use of static code analysis tools coupled with machine learning is used to compare the power of prediction of failure propensity for software quality violations, applying several models for comparing the predictive power of bad smells or possible violations software quality related to the pre-defined metrics in each selected primary study.

3.1.12 RQ4: What would be the contributions?

The collected results indicate a tendency to use static code analysis tools that can be calculated by metrics, use the practice of prioritizing bad smells as described in Table 6, the use of analysis tools static code combined with machine learning is used to compare the power of prediction of failures of software quality violations according to the results present in Table 8 where the main contributions of the selected primary studies are classified in Process (34.28% 12/35), Method (31.42% 11/35), Model (28.57% 10/35), Metric (2.85% 1/35) and Tool (2.85% 1/35). We can affirm that the results are directly linked to the results present in the subsection 3.1.13, since a classification of contribution adopted in its great majority by the selected primary studies are linked to the links that propose a solution to a given problem, whether it is a new solution or a significant reference from previous studies. Petersen et al. (PETERSEN; VAKKALANKA; KUZNIARZ, 2015) highlight small examples are typically used to demonstrate the potential benefits and the applicability of the proposed solution.

3.1.13 RQ5: What research methods were used?

Table 9 shows the relation between the primary studies and six empirical methods (PETERSEN et al., 2008; WIERINGA et al., 2006). Most studies (48.57%, 17/35) focused on proposing new solutions. This result indicates that the primary studies were chiefly concerned

Table 8: Study classification by contributions (RQ4).

Classification	Amount	Percentage	List of primary studies
Process	12/35	34.28%	[A9], [A17-18], [A23], [A25], [A28], [A29], [A31-35]
Method	11/35	31.42%	[A4], [A10], [A12], [A15-16], [A19-20], [A22], [A24], [A27], [A30]
Model	10/35	28.57%	[A2-3], [A5-6], [A8], [A11], [A13], [A14], [A21], [A26]
Metric	1/35	2.85%	[A1]
Tool	1/35	2.85%	[A7]

with bridging research gaps by proposing techniques to deal with design models. The primary studies predominantly sought to propose a new solution, instead of significantly extending an existing technique. The potential benefits and applicability of these solutions have been demonstrated through small examples or initial empirical studies supported by discussions and implications. Robust and practical studies that brought evidence about the effectiveness of the solutions have not been identified. Case studies in the industry considering context variables have not been reported. This may be indicative of an area still maturing and expanding.

Some studies (25%, 9/35) were classified as validation research, which proposed some new techniques, but have not yet been implemented in practice, being evaluated through empirical studies in laboratories. Müller and Fritz [A28] show through an empirical study that biometrics can be used to predict quality concerns of parts of the code while a developer is working on.

The results indicate that little has been done to discuss the problems identified with prediction techniques. Most studies make only notes for identifying anomalies, security, and vulnerability issues as examples. Finally, the lack of a massive amount of empirical studies may indicate that the evaluation of prediction techniques may be based mainly on experts'reflection, not on empirical evidence.

Table 9: Study classification by research methods (RQ5).

Classification	Amount	Percentage	List of primary studies
Solution Proposal	17/35	48.57%	[A4-9], [A12], [A14-16], [A20], [A23], [A26-27], [A29], [A32], [A35]
Evaluation	9/35	25.71%	[A1-3], [A18-19], [A21-22], [A25], [A34]
Validation	9/35	25.71%	[A10-11], [A13], [A17], [A24], [A30-31], [A33]

3.1.14 RQ6: Where have the studies been published?

This section investigates when and where primary studies were published to accurately pinpoint trends in publication. Figure 3 presents the primary studies chronologically, organizes them by type of publication and shows the number of studies published per year.

Number and venue of publications. The blue dashed line in Figure 3 counts the number of articles published per year. The results indicate that 62.86% (22/35) of the primary studies were published in conferences, while 34.29% (12/35) in journal, showing a predominance of

publications in venues that encourage synchronous discussion by researchers. Based on the premise that articles published in journals are more robust, this may indicate a new or maturing area of research. The publications were more concentrated from 2016 to 2019. Such research on prediction of design problems may have gained momentum for two reasons: (1) the maturation of the research area itself. that brought well-established concepts about catalogs of code anomalies and refactorings, as well as empirical knowledge about how certain code or social characteristics impact the incidence of design problems; and (2) machine learning techniques are being widely explored to solve practical software development problems.

Trends. Although there is not yet a consistent upward trend, the number of published studies has been growing. After the first publication in 2005, four and seven articles were published in 2011 and 2017, respectively, representing the tops reached over the years. This growth is accompanied by strong fluctuations, alternating with periods with a maximum of two published articles (2005 to 2009 and 2013 to 2015) to nine or more published articles (2010 to 2012 and 2016 to 2019). In addition, 2017 stood out with a greater number of articles produced than other years. Articles published in premier conferences and journals, such as SANER, ICSE, ASE, MSR, ICSM, JSS, IST, TOSEM, IEEE TSE, show that robust research has already been carried out. Although many studies have been published, there are still challenges worth exploring, which are discussed in the following section.

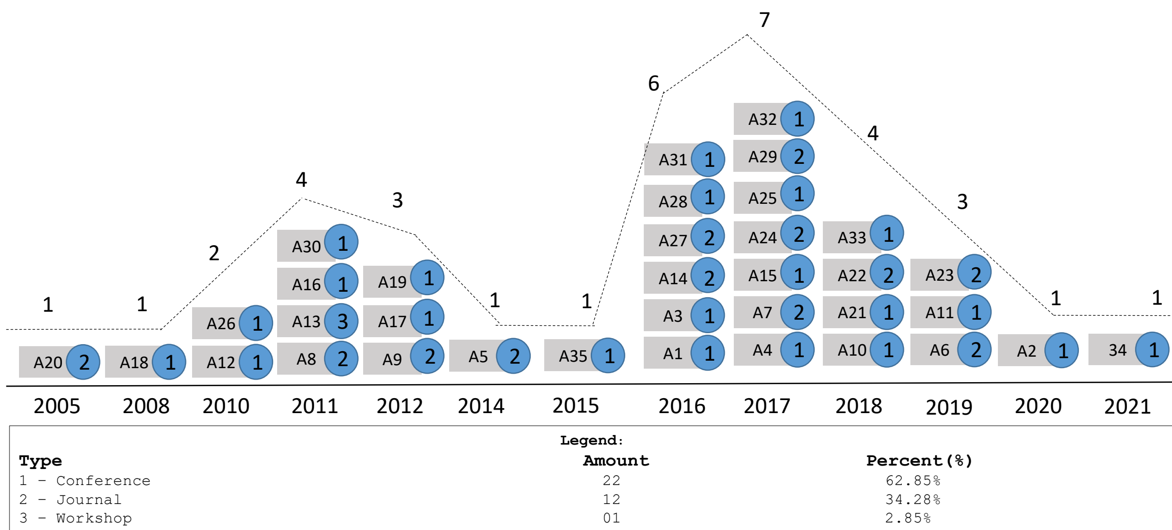

Figure 3: Distribution of the primary studies based the explored research topics over the years.

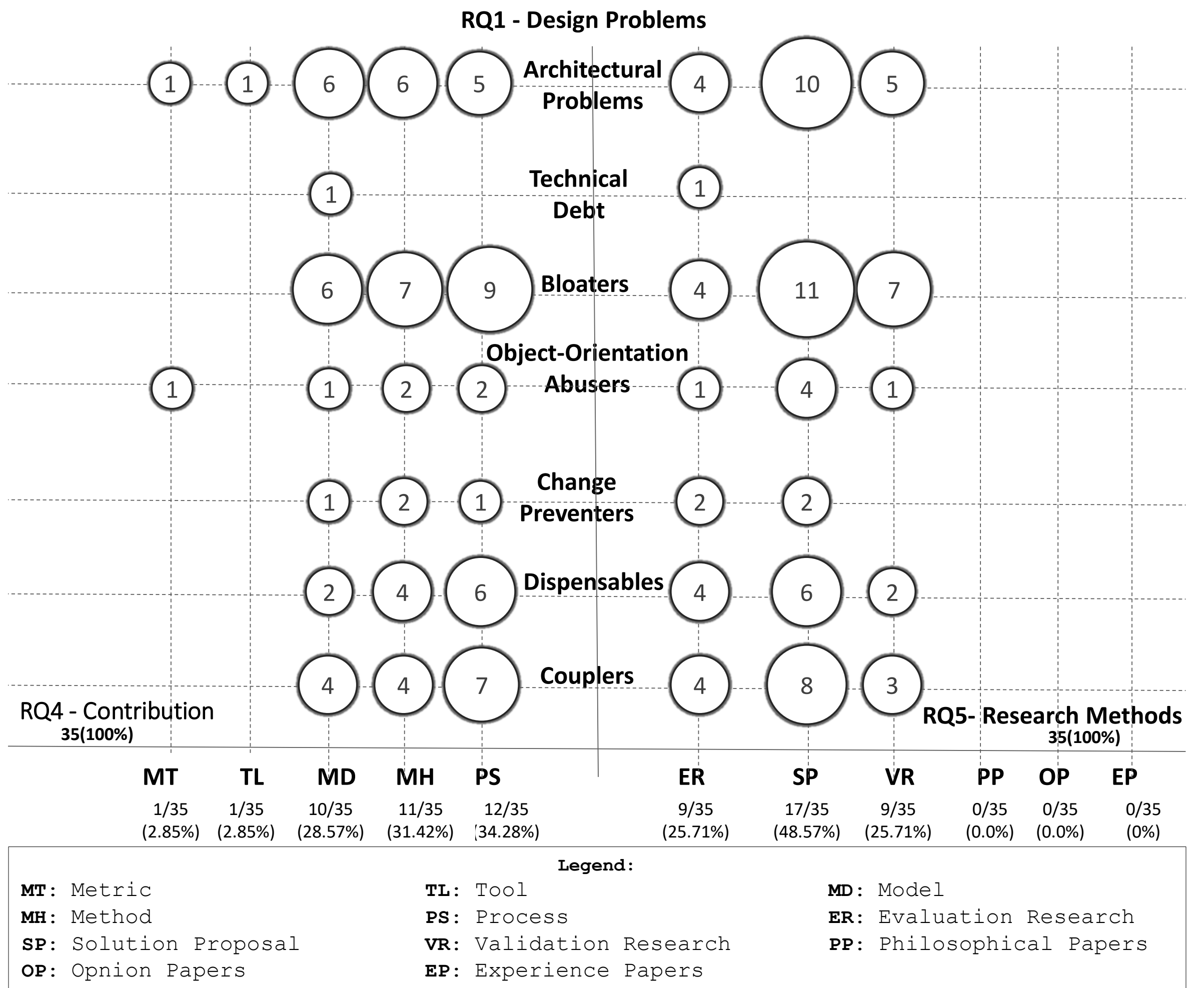

Figure 4: Bubble chart that shows the relationship among three variables.

Table 10: Place of publication of studies (RQ6).

Classification	Amount	Percentage	List of primary studies
Journal	12/35	34.29%	[A5-9], [A14], [A15-19], [A20], [A22-24], [A27], [A29]
Conference	12/35	34.29%	[A1-4], [A10-12], [A21], [A25-26], [A28], [A30-35]
Workshop	12/35	34.29%	[A13]

3.1.15 Discussion and future directions

Discussions about the data collected, seeking to explore the main point of RQ5 and RQ6, that is, reveal where the selected studies are being published over the years through the representation of a graph shown in Figure 4. The classification of selected studies was based on the year of publication, type of publication (workshops, conferences, newspapers, and magazines), and the number of studies published per year. Figure 3 presents the data obtained based on the 35 selected studies, showing quantitatively the results presented by RQ6.

3.1.16 Distribution of primary studies

Figure 4 introduces a bubble chart that organizes primary studies in three dimensions (d1, d2, d3), where d1 represents the main contributions (RQ4), d2 is the adopted research method (RQ5), and d3 is the explored design problems (RQ1). Each bubble has values assigned to d1, d2, and d3. This bubble chart helps grasp relations among the main contributions (RQ4), the research methods (RQ5) and the design problems (RQ1). That is, it shows how primary studies have made a triangulation between RQ1, RQ4 and RQ5.

It is observed that prediction techniques for source code design, even though it is a recent research area, have many studies published since 2014 and continue to grow. This result shows that this area of research has been very active in recent years. After identifying the type of publication of these studies, it is revealed that the researchers who contributed most to the subject made their publications in recent years at conferences, represented by a total of 51.35% of the selected studies.

The results did not present statistical qualifiers and were not compared with other results studied since research on this topic has not been developed by other researchers previously. Some recommendations for future research would be: increase the breadth of analysis of selected studies, refine research on fundamental software quality issues; conduct a systematic review literature to examine best practices related to source code analysis approaches, technologies or tools, comparative analysis information; Also, this work may be the first step towards an ambitious agenda on how to advance the current literature on techniques for predicting source code design problems.

3.1.17 Future challenges

(1) Good quality management in software projects. It is important to identify the main software quality guides most used in the current market. The search for quality to meet customer needs is no longer a differential competitive, but an obligation for any business to survive in the market. The increase in quality in a company generates positive effects on the company's processes, management, customer service, and strategic planning. Therefore, it is imperative to know which quality tools will provide an effective and clear improvement in software projects.

(2) How to quantify software metrics and their quality. Learning analysis appears as a possibility to address this challenge, recognize the difficulties in generating quantitative security, vulnerability, and design metrics. It will be possible to quantify the impacts on the final quality of the software. It is not easy to quantify the maintainability of software. This measure's primary metric is the time spent on maintenance, considering the time of recognition of the problem, analysis of the problem, specification of changes, modification, tests, and the total time. Current studies that explore the granularity related to syntactic, structural, and semantic similarities are still scarce.

(3) How to extract critical features for knowledge discovery. Machine learning is essential for predicting source-code problems. Training machine learning models demand well-designed datasets. The construction of a dataset is challenging due to the various sources and lack of structured data. Moreover, source code only may not be sufficient to obtain good results. Therefore, another challenge is how to consider the developers' experience in the training process of the machine learning models. Current literature fails to deal with these challenges, leading to great researching opportunities.

3.1.18 Threats to validity

The validity of the results achieved in the systematic mapping depends on some factors present in its structure. The main threats to the validation of this study and the factors used to mitigate them are presented and analyzed:

Selection and quality of primary studies: To guarantee an impartial and comprehensive systematic mapping process and the quality of studies considered relevant, research questions, inclusion criteria, and exclusion criteria were defined by a group of researchers.

The researchers and responsibilities: To review the process of carrying out systematic mapping, conducted by the master student, and to clarify his doubts, while he performed the data extraction process. In this way, studies with a broad overview were obtained.

Number of studies selected: To obtain a wide range of results and necessary data, the search for primary studies was carried out in six repositories of widely known scientific studies (IEEE Explorer, ACM Digital Library, Scopus, Science Direct, Scopus and Google Scholar).

Possibility of a relevant study to be ignored: Although it is plausible that possible relevant studies were ignored in the survey of primary studies, we opted only to read the abstract, title, and keywords in the application of the criteria inclusion and exclusion. However, in step 05, manual search procedures were performed using snowballing techniques to find possible relevant studies in the references of the studies selected in the previous step.

3.2 Analysis of the Literature on Domain-Specific Languages for Specifying Bad Smells

This section presents the analysis of the literature on domain-specific languages for specifying bad smells, described in Section 4.2, SmellDSL, a proposal for a domain-specific language to specify bad smells is implemented. SmellDSL was defined from design decisions made based on three assumptions that improve the specification of bad smells. SmellDSL benefits developers when developing notations bad smells and rules to define them. Support tool, it was a SmellDSL tool, as a tool an Eclipse Platform plugin.

Related works were identified in digital repositories, such as Google Scholar, Scopus (Elsevier) and arXiv, by applying the search string "DSL AND BAD SMELLS". In total, 05 (five) works were selected.

3.2.1 Analysis of related works

(BETTINI et al., 2022). This article presents the new version of Edelta. Several improvements and new features in the compiler and IDE were implemented, thus providing a development environment for real-time metamodel evolution. Edelta 2.0 is supported by an Eclipse-based IDE. Your DSL focuses on a lot of static checks to catch most problems during program compilation. Your DSL was implemented with Xtext, a popular Eclipse framework for developing programming languages and DSL. Xtext, in addition to generating the infrastructure for the compiler, also generates complete Eclipse-based IDE support. This work proposes a meta model that can still be evolved in a code refactoring context, The *SmellDSL* is different because it is a DSL and not a metamodel as proposed, the meta-model does not identify the possible impacts on the understanding of *bad smells*, in wrong encoding.

(RAJKOVIC; ENOIU, 2022). This work presents NALABS, a *desktop* application that relies on .NET standards and packages. The tool was developed in C# for the Windows operating system and has three layers: (i) the pre-processing of requirements documents stored as excel spreadsheets, (ii) the configuration and application of *bad smell* metrics, and (iii) presents the results to the user. As it is an experiment still in its initial phase, it is proposed as future activities to focus on exploring and proposing new specifications of *bad smells*, combining the existing specifications into a single quality and complexity index. This work tries to identify possible problem specifications in natural language but based on metrics that do not make clear their understanding of the problem specification.

(BARRIGA et al., 2021). In this article, an extension called PARMOREL is presented to support *bad smells* detection. The approach is capable of selectively removing *smells* that impact user-defined quality. For this, PARMOREL is integrated with a tool that allows modelers to identify *smells* and refactor them. This extension is based on the integration of tools like Edelta, allowing a model-based assessment. Currently, PARMOREL is limited to quantitative user preferences and needs to obtain a set of actions to modify the support model for detecting *smells*. This work does not specify the impacts of the proposed refactoring, as well as quantitatively limits the user's actions as it needs to obtain a set of actions to modify the proposed model, unlike *SmellDSL* for specifying the *bad smells*, as well as its variations according to each software project and user definitions.

(RWEMALIKA et al., 2021). The purpose of this article is to identify *smells* that occur in SUIT (*System User Interactive Tests*). For this, a literature review was carried out and specific smells of SUIT were identified. This process led to a catalog of 35 SUIT-specific *smells*. Then, an empirical analysis was performed to assess the prevalence and refactoring of this *smells* in 48 industrial test suites and 12 open source projects. It is shown that the same type of *smell* tends to appear in industrial and open source projects, but the symptoms are not treated in the same way. In addition to using metrics to identify *smells*, it does not make clear the impacts that it may cause on the project, as well as determining where maintenance should be performed.

(WŁODARSKI et al., 2019). This article describes the effort to modernize the system of a central bank (CBS) of the corporate branch of mBank. In this effort, the decision was made to first improve the quality of the code base and then start the modernization work. With this initial quality improvement work and use of *bad smells* detectors and testing, frameworks are in continuous daily use. It introduces language-specific *bad smells* detectors that are now run on every software release, giving developers an incentive to continually improve source code quality. This work, by using a catalog of *smells* between 1999 and 2016, may compromise the identification of *bad smells* in your project as well as its classification, as this step must be adapted based on the team dynamics and the context of each project.

3.2.2 Comparative analysis of the selected related works

This Section compares *SmellDSL* with selected studies. Based on criteria (C), serving to identify similarities and differences between the works. The comparative analysis was performed based on criteria, as other works already published (RUBERT; FARIAS, 2022) used this approach, proving to be effective in generating a comparison between the works. By carrying out the comparative analysis, it was possible to identify similar and different points between the proposed *SmellDSL* and the selected related works.

The comparison criteria are presented below:

- **DSL specification for *bad smells* (C01):** this criterion seeks to assess whether the work presents any method of specifying *bad smells* (text, languages, annotations, etc.).
- **Proposes a DSL language (C02):** this criterion aims to identify whether the work proposes a language for specifying *bad smells*.
- **Tool Support (C03):** the work proposes a tool support for the proposed language.
- **Empirical Evaluation (C04):** the work carries out an empirical study to evaluate the benefits of the proposed language together with the tool.
- **Support of *bad smell* (C05):** this criterion seeks to identify whether the works are able to identify the catalogs of the various *bad smells* present in the literature (FOWLER, 2018; SURYANARAYANA; SAMARTHYAM; SHARMA, 2014)
- **Integration with the Eclipse Platform (C06):** This criterion seeks to verify if there is integration with the Eclipse IDE, widely used by the software industry.

Table 11 presents the comparison of works according to defined criteria. It is emphasized that SmellsDSL meets all criteria, highlighting its contributions and limitations. Therefore, this work also identifies as a research opportunity a proposal for a domain-specific language to specify bad smells, which is explored in the Chapter 1, with the research question (RQ4) How to propose a language domain specific for specifying bad smells?.

Table 11: Comparative analysis of related works

Related Work	Comparison criteria					
	C01	C02	C03	C04	C5	C6
SmellDSL	●	●	●	●	●	●
(BETTINI et al., 2022)	○	○	●	●	○	●
(RAJKOVIC; ENOIU, 2022)	○	○	●	●	◐	○
(BARRIGA et al., 2021)	◐	◐	●	●	◐	●
(RWEMALIKA et al., 2021)	○	○	●	●	◐	○
(WŁODARSKI et al., 2019)	○	○	○	●	◐	○

Legend: (●) *Support* (◐) *Partially Support* (○) *Not Support*

4 PROPOSED APPROACH

In this chapter, the proposed SmellGuru approach is presented, which is a machine learning-based approach to predicting design problems. For this, the approach proposes an intelligible workflow (Section 4.1). (Section 4.2) presents a domain-specific language for specification of bad smells. Finally, (Section 4.3) presents details about the implementation aspects of a machine learning model for identification and classification of design problems already cataloged by the current literature.

4.1 Overview of the SmellGuru approach

Figure 5 presents an overview of proposed approach through an intelligible workflow. Note that it is made up of four steps. The initial step starts with data acquisition. In this step, data can be collected in a traditional way with software metrics or with the use of a DSL, detailed in the next section, receiving input data models to define bad smells. In step 2, the activity performed is the identification and analysis of the input data collected to be used in the machine learning model. Initially, the definition of the analysis characteristics of the source code related to bad smells is carried out, each type of characteristic is identified as a possible element of the composition of bad smells already cataloged. Then, the prediction and classification model is executed, with the aid of a set of chosen ML algorithms, it calculates the degree of similarity(S) for each input data of the model present in our database for identification, classification and prediction of possible bad smells. Finally, step four presents a breakdown of the processed data visually. The main contribution of this work is in the initial step 1 of overview, in which we identified the need for a DSL to specify the most relevant characteristics in the definition of bad smells. Each step of the intelligible workflow is described below:

- **Step 1: Data collection.** This step aims to collect data for the integration of steps 2-4, of the proposed approach SmellGuru, has as main objective to collect the data that serve as input parameters for the execution of the classification algorithm of the ML model. This step consists of three essential processes for data collection: definition of the target project; identify project metrics and evaluate information related to project metrics with potential impacts related to design problems. During the execution of this activity and based on the authors' statements already reported in (Section 4.2), there is still no consensus on which metrics to apply to identify design problems and define a bad smell. To explore objective 4 of this work, a domain-specific language is proposed to define bad smells according to the needs of each software project and its possible variations.

  Define target project. In this activity, we select open source projects available on GitHub, choose projects that can be evaluated by supporting tools that measure design quality, and can thus be decomposed for evaluation through reverse engineering. To choose these open source projects, there are some requirements that must be fulfilled,

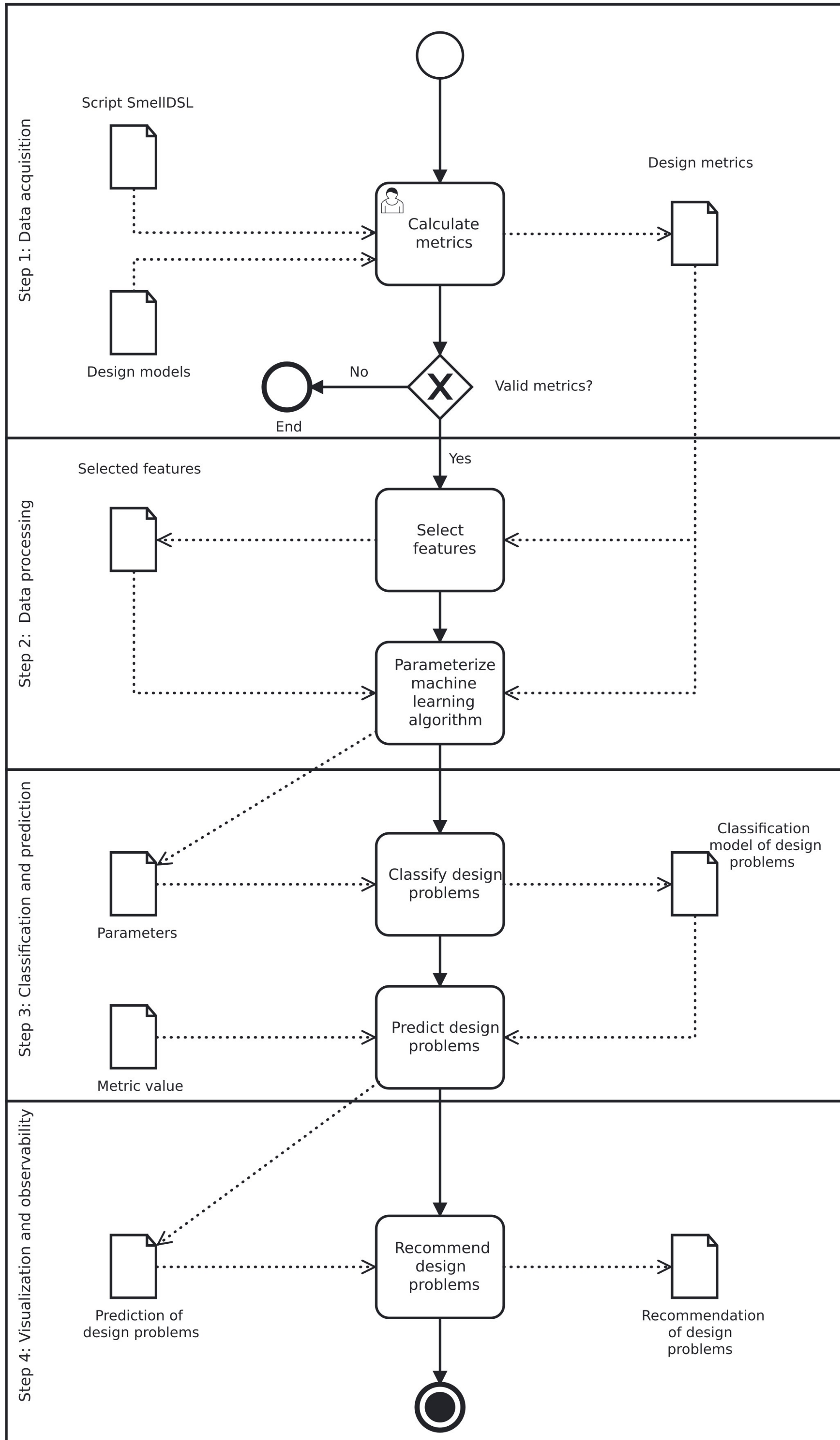

Figure 5: An overview of the proposed approach.

such as: the chosen project must be implemented with development techniques based on object orientation.

Measure design metrics. In this activity, is need the support of a tool that checks design rules to collect software design metrics from your source code. With the support of the Astah UML tool, a reverse engineering tool that produces XMI files from selected project source code, it is possible to collect software metrics. The aim is to ensure that we can visualize the software design ideas for the selected project based on the information obtained through the XMI files which contains information about the project model to be evaluated. From the data generated by this tool, it is necessary to use SDMetrics to identify which classes and methods break design rules. The identification of inconsistencies in the source code is not a simple task, the collected software metrics focus on metrics that characterize the code (Lines of Code - LOC, Number of Attributes - LOA, Number of Methods NOM), complexity metrics (decisions and coupling) such as Weighted Method per Class (WMC), Lack of Cohesion in Methods (LCOM) and finally inheritance metrics such as Depth of Inheritance Tree (DIT) and Number of Children (NOC). Each release of the analyzed project has its metrics collected, and the track of variations and design rule breaks reported are stored for classification of the model. This information is needed as input parameters of the ML model used by the SmellGuru approach.

- **Step 2: Data processing.** The second step of the SmellGuru approach, its objective is to run the ML algorithm using the input parameters provided in the data collection process. Machine learning algorithms typically accept parameters that can be used to control certain properties of the training process and the resulting ML model. This activity is essential for classifying and identifying design problems proposed by the approach.

  Running the machine learning model. In this activity, the approach runs the ML algorithm with the input parameters collected in the previous step. The purpose performs a classification according to the selected project metrics to identify the project problems from the code smells. It needs to run a subroutine in its mainstream data processing. The subroutine is used to transform the data from the algorithm's input parameters into the algorithm's internal data structure. Briefly, the algorithm converts the evaluation resource files of the project to be evaluated according to each selected property as input parameters, based on the lines defined in the evaluation file of the selected project.

- **Step 3: Classification and prediction**

  At this moment, the collected raw data has already been converted into indicators of possible source code anomalies, using machine learning techniques to train, test and validate the data model. These indicators are sent to a storage engine, giving rise to a supposed data set for the next step. Briefly, this step aims to use data from previous releases already analyzed to predict future design problems, through statistics based on the learning of the proposed model.

- **Step 4: Visualization and observability**

  The availability of results is the fourth step of the approach SmellGuru, its objective is to display the results related to the recommendation of possible design problems based on code smells. This step can be composed by generating a step defined as Machine Learning as Service, as can be seen in Figure 5.

  Generate design problems recommendation. The design problem recommendations are reported based on the information provided from the previous step, these data are made available through SmellGuru dashboards, this step is composed of statistical data processing for the provision of results. In this step, the last SmellGuru subroutine is executed, which is responsible for generating the information in the form of indicatives for the possible design problems that were recommended. In this step, SmellGuru introduces which features are part of each design problem and indicates for each functionality which classes and methods may possibly have problems at the design stage before they are committed to the source code. Basically, the result available in a panel, lists the recommended design problems, the names of the features that are part of that smell, and the classes and methods that are part of each affected functionality. Through this final result that the user can initiate an action to avoid potential design problems from the beginning. Predict relevant design qualities such as failure proneness or maintainability to better focus your review and testing efforts based on reported data, so you can find failures sooner and save development costs.

4.1.1 Component-based architecture

Figure 6 presents a component-based architecture to support the implementation of the SmellGuru approach. Together, the components are responsible for implementing each step of the proposal process in Figure 5. For a better understanding of the proposed architecture, each module is described in an abstract point of view, in relation to their behavior rather than the technology applied as follows:

- **Data connector:** This component allows for integration between systems and application compatibility. Web Services allow for integration between systems and application compatibility. Thus, new applications can efficiently interact with those that already exist, and systems developed on different platforms are compatible. So new applications can efficiently interact with those that already exist on different platforms to make them compatible, the connector receives data through Web Services that use standard protocols such as HTTP, XML, and SOAP (Simple Object Access). This communication layer can analyze data in a high-level format such as JavaScript Object Notation (JSON), or Extensible Markup Language (XML). A Web Service in JSON format is developed for the simulation of values. As for the confirmation of the HTTP request data: in the POST,

the values sent and a message of sending success are returned (in JSON format); in GET, values are returned according to the standard established by the protocol to be used by the approach;

- **Data adapter:** Provides diagnostic format interaction and evaluation between the Data Connector and Data Processor components to reduce architectural friction in the case of the Data Connector change. Python supports handling JSON native. An alternative is to use one of your libraries that parses a string in JSON, or optionally makes an HTTP call using a component that evaluates a JSON string returned by this call according to the needed parameters of validation for the data in this approach composition step;

- **Data processor:** Data transfer between applications is done through an API — Application Programming Interface — which, among other formats, uses JSON notation to structure the information transmitted. The simplicity with which the data is structured in the JSON format allows it to be used in any type of programming language.

- **Machine learning model:** This component abstracts information according to the machine learning model already trained and evaluated for the classification of data in the process. At this point, artificial intelligence techniques are introduced to evaluate the collected data and classify the smells found. This analysis can be performed by a machine learning model or tool, which the behavior and interests of these data, generating a pattern with information about the types of smells found or classified. From these models, recommendations are made to support the prediction of smells. One of our challenges is, through artificial intelligence, to find patterns and similarities between different information artifacts to increase the efficiency of smell prediction.

- **Data server:** Acts as the main component of the architecture. Receive data from Data Connector, processes using the Data Processor, and sends it to be displayed by the SmellGuru. This component also receives information from the trained machine learning model, relating artifacts to their metrics and persisting them across the database engine for data evaluation process.

- **IDE plug-in:** Communication with the data server is your main responsibility, the handling and presentation of changed data within the IDE will be just a panel, with the objective of providing better visualization of the data capable of allowing insights into the presented data and related bad smells parts. The plug-in's goal is to simplify the data collection process and enable rapid development of integrated machine learning solutions.

Software artifacts information that is processed and evaluated by the approach will not be stored at any time during processing, information issues pertaining to the software project artifacts will be encrypted and later discarded after the dashboard is presented. Faced with this problem, the approach can be adapted to store and group project information to be evaluated. In

this way, the information about possible problems in the software project is preserved, guaranteeing the privacy of the project available. After presenting an overview of the proposed process Section 4.1 and presenting the architectural components, the following section discusses the implementation aspects of the SmellGuru approach.

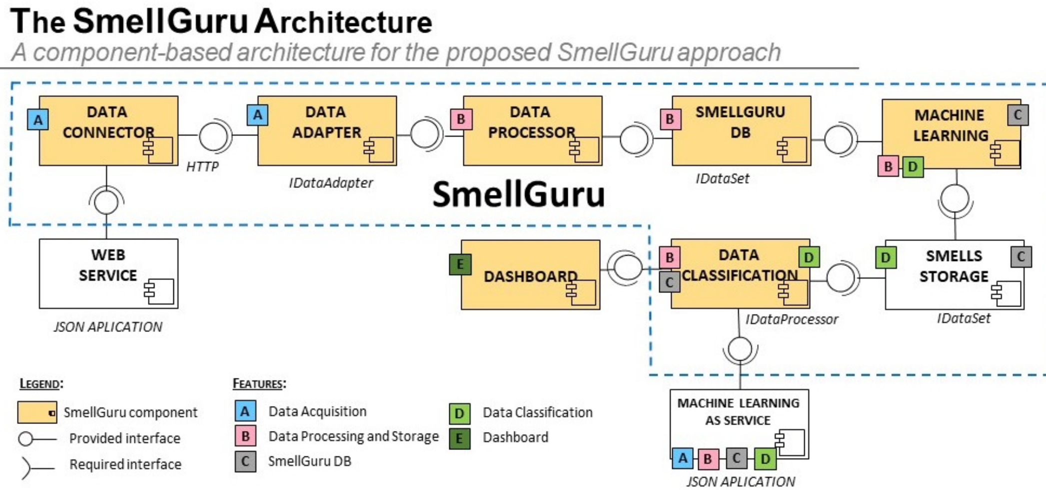

Figure 6: A component-based architecture for the proposed SmellGuru approach

4.2 Domain-Specific Language for Specification of Bad Smells

This Section introduces SmellDSL, which is a language, domain-specific specific language for specification of bad smells. The SmellDSL was defined from project decisions taken based on three premises that enhance the specification of bad smells. SmellDSL benefits developers by introducing notations to define bad smells and rules to identify them. A support tool, SmellDSL tool, was implemented as an Eclipse Platform plugin. An exploratory empirical study was carried out with 12 participants, who used the SmellDSL tool to specify 8 bad smells, generating 96 evaluation scenarios.

4.2.1 Language Design Decisions

SmellDSL was designed with some assumptions in mind: (1) it supports the specification of bad smells already cataloged (in an informal way) in the literature, enhancing wider adoption; (2) uses notation and concepts found in object-oriented languages, facilitating the adoption of developers already familiar with object-oriented languages; and (3) separate the definition of bad smells characteristics and the rules for identifying them, allowing bad smells to be identified differently by different development teams. For example, a class with 50 methods and 30 variables can be considered as *God Class* for one team, while not for another. In the following paragraphs, language design decisions are presented.

Allow to specify the bad smells cataloged in the literature. The language was designed in such a way that developers could specify the bad smells already cataloged in the literature (FOWLER, 2018; SURYANARAYANA; SAMARTHYAM; SHARMA, 2014; ALKHARABSHEH et al., 2019), as well as how to specify new bad smells. For this, the language project sought to contemplate the main elements commonly used in the specification and characterization of bad smells. Examples of such elements would be: features would be attributes that characterize the bad smell; symptoms, perceptions or consequences generated by the occurrence of a bad smell; and treatments, recommendations for actions to mitigate unwanted symptom

Use of reuse concepts found in object-oriented languages The language brings the concept of inheritance to allow bad smells to share features, symptoms, and treatments, as well as establish a semantic relationship of the type "is a ". Inheritance (or generalization) is a taxonomic relationship between a more generic bad smell and a more specific bad smell. Each instance of the specific bad smell will also imply the occurrence of the generic bad smell. The more specific bad smell inherits the characteristics of the more generic bad smell. The inheritance relationship is owned by the more specific bad smell.

Separate the definition of the characteristics of bad smells and the rules for their occurrence. The bad smells cataloged in the literature are specified informally. Researchers and professionals end up having difficulty understanding the characteristics of a bad smell, including its properties (or metrics), symptoms, and treatments, as well as the logical rules that guide the occurrence of bad smell. SmellDSL seeks to exactly mitigate this problem by allowing a separation between the definition of a bad smell, including its properties, symptoms, and treatments, and the specification of rules for the occurrence of bad smells. For example, the literature defines smell *God Class* (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014; FOWLER, 2018; PALOMBA et al., 2014) as being a class that contains many variables, methods, or lines of code. However, the clarity of this definition is somewhat questionable, especially considering the elements that characterize bad smells, as well as the criteria to objectively define their occurrence.

4.2.2 Language Grammar

The grammar of SmellDSL was defined using BNF notation and incorporated into the Xtext framework. Figure 7 represents the *railroad* syntax of the main elements of SmellDSL. The language has three core elements: *smelltype*, smell and *rule*. A smelltype is an abstract type of bad smell that cannot be detected. That is, there are no rules to detect them, being used only to express a concept. On the other hand, a smell is a concrete type of design problem that can be detected. That is, it is possible to define a rule to detect a bad smell. Thus, each instance of a smelltype is an instance of smell (concrete subtype). An abstract type may provide no implementation or incomplete implementation. A smell is composed of one (or more) *feature*, a *symptom* and a *treatment*. A feature represents a measurable feature used to detect a smell.

Each feature must have at least one *threshold*, which defines a measurable measure reached by the feature that requires attention. For example, a feature could be the number of methods in a class with limits of 50 and 80, where 50 would indicate a point of attention and 80 the need for immediate refactoring. A smell also has a symptom and a treatment, which represent the code's perceptions of something unwanted and actions to reduce such unwanted perceptions, respectively.

Figure 8 presents a code example, representing grammar sentences. smell *LongMethod* is defined as a subtype of *smelltype Bloaters*. *LongMethod* has three features, *nlin*, *comp* and *nparm* , with three limits each. The *rule rule* 3 indicates that when the *featuyre LongMethod.nlin* is greater than the *threshold nlin.maior100* or the *feature LongMethod.comp* is equal to *comp. Baixo*, then something must be done, which is represented by the message "Reevaluate the encoding due to its complexity".

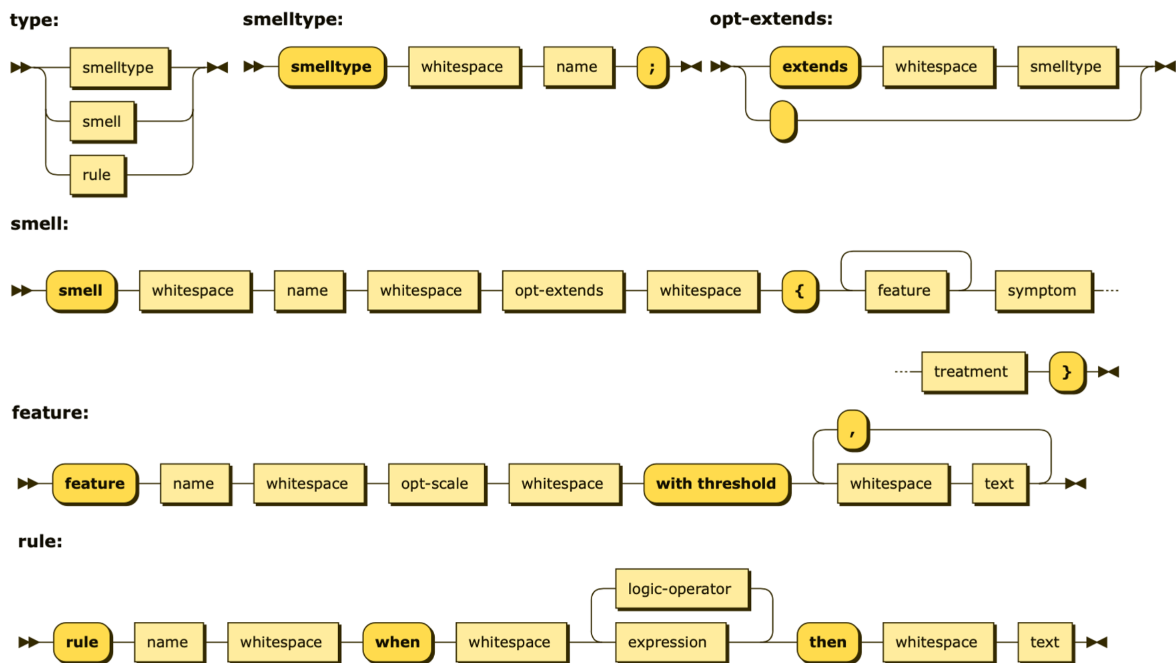

Figure 7: Diagram of *railroad* of the syntax of the language.

4.2.3 Implementation Aspects

This Section proposes the SmellDSL tool, which is a support tool for the proposed language. It is integrated into the Eclipse platform and implemented using the Xtext [1] framework. Xtext is a framework for developing programming languages and domain-specific languages. The SmellDSL tool offers a set of resources to facilitate the elaboration of the code that defines the bad smells, as well as the specification of their occurrence rules represented in Figure 9. The main features or characteristics of the SmellDSL tool would be:

[1]XText framework: http://www.eclipse.org/Xtext

```
smelltype Bloaters
smell LongMethod extends Bloaters{
    feature nlin is Nominal with threshold menor50, entre50e100,maior100
    feature comp is Nominal with threshold Alto, Medio, Baixo
    feature nparm is Interval with threshold dez,vinte, trinta

    symptom ObjetoMuitoGrande
    treatment QuebrarObjetoPartes
}
rule regra1 when LongMethod.nlin <= nlin.menor50
then Nenhuma acao necessaria

rule regra2 when LongMethod.nlin <= nlin.menor50 AND LongMethod.comp == comp.Alto
then Reavaliar a codificacao

rule regra3 when LongMethod.nlin > nlin.maior100 OR LongMethod.comp == comp.Baixo
then Reavaliar a codificacao devido a sua complexidade
```

Figure 8: Code example of SmellDSL.

- **Syntax Coloring:** keywords, comments, strings, and other basic language elements are colored according to the syntax definition.
- **Error checking:** the tool identifies and marks errors in the elaborated code. Examples of errors would be language syntax violations or references to undefined elements.
- **Autocomplete:** This feature allows autocompletion, making it easier to specify bad smells.
- **Formatting:** Code formatting rules (eg line breaks, spacing, and tabs) can be defined and automatically applied to code.
- **Rename refactoring:** When you rename a code element, all its references are automatically updated.
- **Proposed quick fixes:** Problems found by the bug checker can be resolved through proposed automatic fixes.
- **Information when hovering the cursor:** Displays additional information when the mouse hovers over a specific language element, for example the comments associated with that element.

4.3 Machine Learning Model for Predicting Design Problems

This Section explores **Step 3: Classification and Prediction.** It is important to point out that the steps at this moment are not discussed with SmellDSL. Unfortunately, bad smells are typically informally defined and specified, hampering understanding and refactoring tasks. Data processing is transversal in several domains related to design problems, and requires techniques for analyzing such data. In this sense, machine learning was used to offer us a new option of tools to support decision making in identifying bad smells.

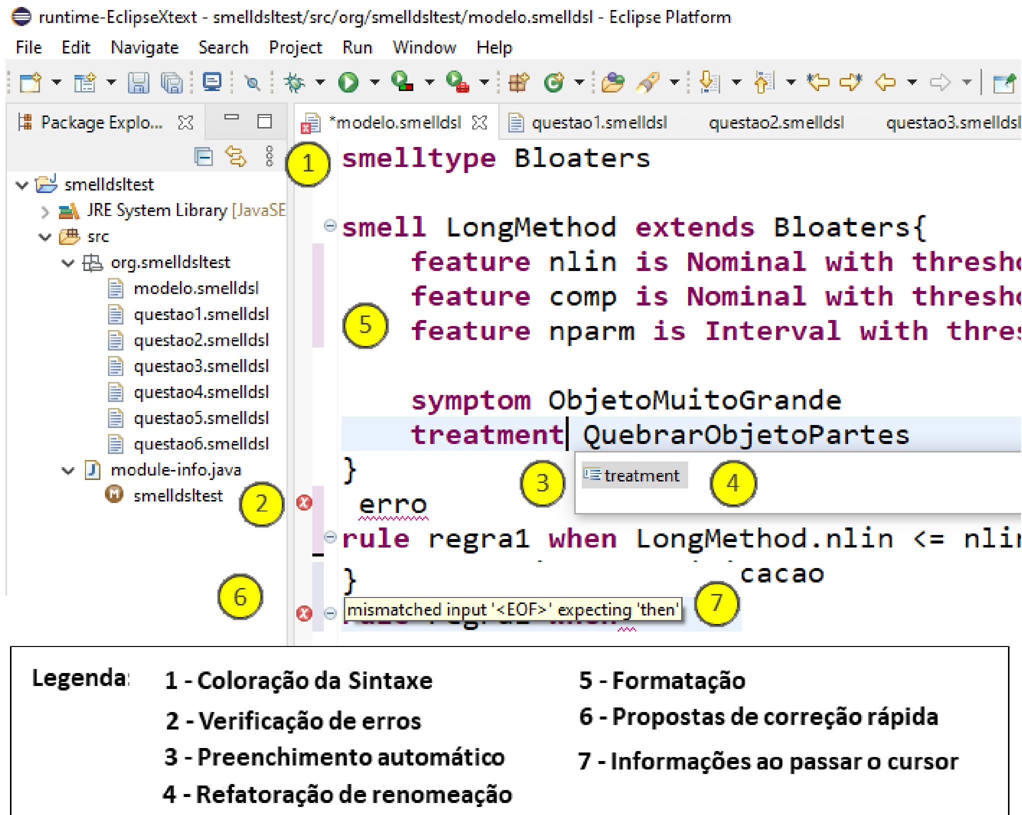

Figure 9: SmellDSL tool integrated into the Eclipse platform.

There are two main tasks that machine learning problems fall into: predictive and descriptive. In predictive tasks, the goal is to find a function, also called a model or hypothesis, from training data. Such data is described by the input attributes. The possible label values form a set that defines the output attribute of the function. Such an output attribute is commonly known as a class. In descriptive tasks, the goal is to explore or describe a set of data. In general, in this type of problem, there is no class attribute, that is, the data was not labeled. Thus, the learning algorithms used in these tasks do not use the class attribute and, therefore, follow the unsupervised learning approach.

In this section, the focus is on supervised learning only. With that, the objective of this section is to present some scenarios of using machine learning to forecast demand in the identification of design problems. Is describe how the assessment of the knowledge extracted by the learning algorithms is carried out, as well as the tools used in the demonstration of the assessment process.

4.3.1 Methodology

The study is guided by tree research questions (RQs).

RQ1: Is the design impacted due to the presence of code smells? Evaluate the impact in terms of process and technical based on metrics that can identify possible code smells that

influence Design? Process, social and technical aspects may be preventing or amplifying design degradation. To capture these aspects, we used a set of metrics detailed in Section III - RQ 1 aims to investigate which metrics can distinguish between projecting impact and non-impacting changes.

RQ2: How well do ML algorithms perform in predicting design-impacting changes? As soon as we show empirical evidence that distinguishes impact and non-impacting changes, RQ2 aims to investigate the use of supervised ML Techniques to help those involved in the software project to automatically make notes and their decisions.

In practice, some prediction algorithms perform better than others, depending on the task. Thus, we compared the performance of two interpreter ML algorithms: SVM and Random Forest. We chose these algorithms since they provide an intuitive and easy-to-explain model (SHARMA; MISHRA; TIWARI, 2016; KOTSIANTIS; ZAHARAKIS; PINTELAS, 2006). To this end, we apply the ML algorithm using two sets of resources: one set using only process resources, one set using only the technicians. By answering RQ2, we will be able to identify which types of resources are the best predictor, as well as the effectiveness of the combination of process and technical characteristics. In addition, we also assessed the effectiveness of a selection step feature for both sets.

RQ3: What features are the best indicators of change that impact design? RQ3 aims to underestimate which characteristics are considered the most relevant by the models. Such knowledge is essential because, in practice, a model should be as simple as possible and require as little data as possible. By answering RQ3, we will be able to provide insights to professionals and researchers as to the factors that best indicate impact changes in the project.

4.3.2 Classifier - Random Forest Algorithm

In this implementation step, the Random Forest classifier is selected, for training our prediction model, as it is a statistical method of supervised learning, which can be used in classification problems and in making predictions. From an existing dataset, the method creates a representation of the knowledge embedded therein, in tree format.

There are two main reasons for using these methods in Random Forest, firstly they increase the accuracy of the results by decreasing the correlation of each tree, and, in addition, they provide estimates of the generalization errors of the trees on a continuous basis.

In this algorithm, a large number of decision trees are built as they operate together. Decision trees act as pillars in this algorithm. Random forest is defined as the group of decision trees whose nodes are defined at the pre-processing step (BOUAZIZ et al., 2014). After constructing multiple trees, the best feature is selected from the random subset of features (BREIMAN, 2001; KABIR et al., 2021). To generate a decision tree is another concept that is formed using a decision tree algorithm. So, a random forest consists of these trees which are used to classify a new object from the input vector. Each decision tree built is used for classification in (SHAH

et al., 2020).

4.3.3 Predictive model

The literature covers a wide variety of ML techniques, each with specific characteristics and applications. The focus of this section is the application of these models for the predictive identification of design smells. We intend to highlight the commonly used ML techniques and the reasons for selecting specific techniques.

Predictive models are usually data-driven models that require a variety of data streams provided by multiple real-time and offline sources. Data also comes from Computerized Maintenance Management Systems (CMMS). Failure-related data is also necessary to build and test the predictive models (DALZOCHIO et al., 2020).

The works present architectures, strategies, principles, and tools that seek each step of the systems implementation process to allow predictive maintenance (BOUSDEKIS et al., 2019; ANSARI; GLAWAR; SIHN, 2020; SARAZIN et al., 2019; HEGEDŰS; VARGA; MOLDOVÁN, 2018).

Several authors use Artificial Neural Networks (ANN) to tackle problems related to predictive maintenance. (LI; WANG; WANG, 2017) proposed a framework for fault detection and prediction, which is capable of performing error correction regardless of machine or process type.

Other solutions involve the implementation of Recurrent Neural Networks (RNN) which are a type of ANN capable of incorporating memory. (RIVAS et al., 2019) adopted Long Short-Term Memory (LSTM) RNN model for failure prediction. The authors focused on creating an LSTM model to identify a possible future malfunction using two models. (CACHADA et al., 2018) also used LSTM along with a second technique called Gated Recurrent Unit (GRU) for a similar purpose. Both models were applied because they implement the ability to consider historical data to predict future behavior. Several authors (SCHMIDT; WANG, 2018; ZHOU; THAM, 2018) assess the performance of ANN and compare it with other techniques like Support Vector Machine (SVM) and Randon Forest (RF). Yet in the area of ANN, some works consider the implementation of Auto-Associative Neural Networks (AANN).

4.3.4 Description of Dataset

The dataset referring to the design smells was obtained with the support of the tool SDMetrics. The Table 12 presents the design metrics used in our ML training model. To speed up the data collection process due to the unavailability of open source models for the software, the authors used a reverse engineering Tool (Reengineer) if necessary to be compatible with SD-Metric software quality tools, which is an optional step. The software quality analysis tool uses XMI file type. During the execution of step 1, the SDMetrics tool is used to support data col-

lection. This tool works with all UML design tools that support XMI. By using object-oriented metrics of project size, coupling and complexity, in addition to establishing quality benchmarks to identify potential design problems from the beginning of software production, it has become a necessary tool for the other steps of the approach to be executed.

Table 12: Description of Class level Design Metrics with categorization.

Metric	Category	Description
NumOps	Size	The number of operations in a class.
NumPubOps	Size	The number of public operations in a class.
Setters	Size	The number of operations with a name starting with 'set'.
Getters	Size	The number of operations with a name starting with 'get', 'is', or 'has'.
Nesting	Outros	The nesting level of the class (for inner classes).
NOC	Inheritance	The number of children of the class (UML Generalization).
NumDesc	Inheritance	The number of descendents of the class (UML Generalization).
NumAnc	Inheritance	The number of ancestors of the class.
DIT	Inheritance	The depth of the class in the inheritance hierarchy.
CLD	Inheritance	Class to leaf depth.
OpsInh	Inheritance	The number of inherited operations.
AttrInh	Inheritance	The number of inherited attributes.
NumAssEl$_{s}sc$	Coupling	The number of associated elements in the same scope as the class.
NumAssEl$_{s}b$	Coupling	The number of associated elements in the same scope branch as the class.
NumAssEl$_{n}sb$	Coupling	The number of associated elements not in the same scope branch as the class.
IC$_{A}ttr$	Coupling (import)	The number of attributes in the class having another class or interface as their type.
EC$_{P}ar$	Coupling (export)	The number of times the class is externally used as parameter type.
IC$_{P}ar$	Coupling (import)	The number of parameters in the class having another class or interface as their type.
Dep$_{O}ut$	Size	The number of dependencies where de class is the client.
Dep$_{I}n$	Size	The number of dependencies where the class is the supplier.

4.3.5 Implementation Aspects

This section aims to present the implementation aspects, reporting the decisions taken to enable the development of prototype 1. Section 4.3.6 of the proposed approach presents an initial proposal from the SmellGuru dashboard discussing how the main architectural components were implemented in terms of the technology used. Section 4.3.7 introduces the extension of the approach prediction model.

4.3.6 A Proposal of SmellGuru Dashboard

Figure 10 presents an initial proposal for a smells dashboard. The metrics of possible code smells with greater relevance in the analysis will be presented randomly to simulate the real data collected by SmellGuru through integration via Web Services, information regarding the

prediction of smells will be presented, according to the features sent and represented by the real state of any software project. The approach was adopted for illustrative purposes only, as the main objective of this study is to assess the tool's acceptance and the impact of presenting on code smell prediction.

The SmellGuru Data Connector component (Figure 6) was performed using the WSDL, which stands for Web Services Description Language, it is an industry standard to describe Web Services to eliminate as much as possible the need for communication between the parties involved in data integration, the documentation tag allows us to include our documentation, dispensing with an auxiliary document, to explain the objectives of the Web Service or what each field is for. This not only ensures that information is centralized, but also makes it easier for others to read. The data server, that is, Smell Server, was developed using Anaconda to simplify the management and deployment of packages. The distribution includes data science packages suitable for Windows, Linux, and macOS.

All transactions between IDE and the data server were performed through JSON and CSV over hypertext transport protocol (HTTP), managed by Python's JSON library and CSV. As a storage mechanism for processed data, MYSQL Workbench, a once series database, was adopted. This type of database allows for a high rate of insertion operation, in addition to the analysis of fluctuating values over time. The data stored in MYSQL was consumed by the Power Bi dashboard engine, which was responsible for presenting the processed data in an easy-to-use way (Figure 10). This choice was made over the native integration between Power Bi and MYSQL.

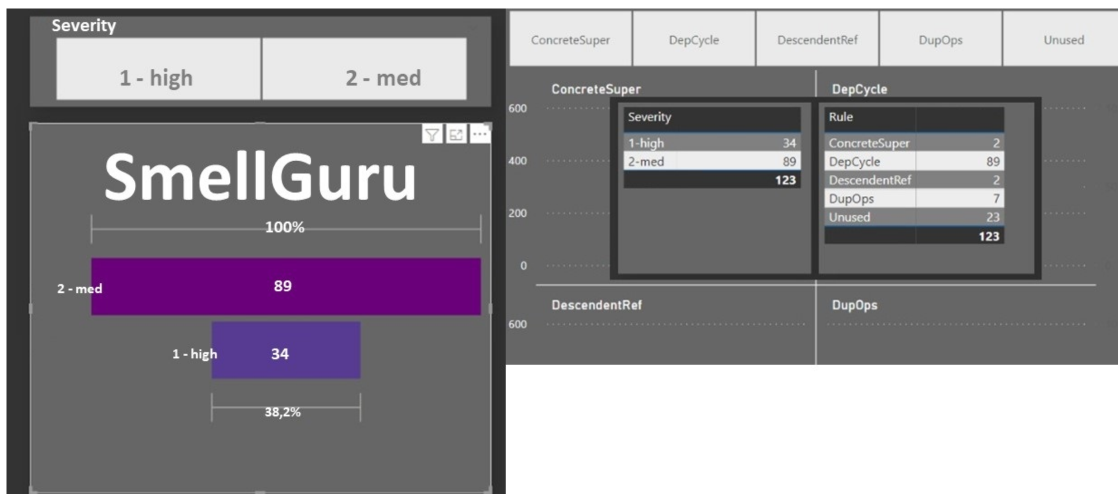

Figure 10: Visualization SmellGuru

4.3.7 An extension of the approach

SmellGuru evaluated data prediction, in a machine learning model, made available by our collaborators, in Python, that is, after data processing, they will be categorized and grouped

according to the smell characteristics to be classified. In this way, the data is contextualized, providing visual feedback on the control panel. The SmellGuru prediction approach used the Anaconda Python code editor, which provides a high-level API for development (or plug-ins). The tests were all run using the *Jupyter Notebook* [2] environment with *Python* version 3.7.7 and manipulate the dataset using the *Python Data Analysis Library* [3]. We generate the graphics using the *matplotlibrary* [4]. The learning algorithm selected for the classification task was RF, and the SVM was selected for comparison. The implementation of these algorithms is provided by the library *scikit-learnig* [5] using version 0.23.1.

Data collection with artifact metrics tools: For database consolidation in this step we use three tools to collect artifact metrics:

(1) *SonarQube* [6] has GitHub integration and raises issues whenever a piece of your code breaks a coding rule, be an error that will break your code (bug), a point in your open source to attack (vulnerability) or a maintenance issue (code smell). To make sure that SonarQube results are relevant, it is necessary to narrow the focus or configure what to analyze for each project.

(2) *Astah* [7] is a modeling and diagramming software, helps engineers and developers plan, create, communicate, and understand. The software was used for the integration of artifacts (source code), reverse engineer data from models to learn the impact of development in the real world.

(3) *SDMetrics* [8] was used to collect information from artifacts related to design rules, this tool automatically detects incomplete designs, incorrect, redundant or inconsistent, styling issues like circular dependencies, violating naming conventions, and so on.

4.3.8 Experimental Design

This phase was characterized by: (1) a selection of GitHub projects, real software to be adopted by the approach SmellGuru, (2) the execution of the experimental process, and (3) the post-experiment data collection. The selected software projects must use development criteria based on object orientation, in different programming languages, were selected for convenience and ease of access. For the execution of the experimental process, the following steps were taken:

- **Step 1:** GHTorrent information collection and evaluation using the SDMetrics tool, design quality measurement, analyzes the structure of your UML models;

- **Step 2:** Building a Prediction Model, prediction models try to estimate the future qual-

[2] https://jupyter.org/
[3] https://pandas.pydata.org/
[4] https://matplotlib.org/
[5] https://scikit-learn.org/stable/
[6] https://www.sonarqube.org/
[7] https://astah.net/
[8] https://www.sdmetrics.com/

ity of a system from internal quality attributes that are measurable at present. This is achieved by empirically exploring the relationships between internal and external quality from systems developed in the past, and applying these findings to new systems. In the following, we describe how to build and use a prediction model for class fault-proneness from the structural properties of a class. Figure 11 depicts the steps involved in building the prediction model;

- **Step 3:** Perform an analysis based on your own perception;
- **Step 4:** In Python, the information for the code will be consumed beforehand. implemented and interpreted, present the metrics based on your own perceptions;

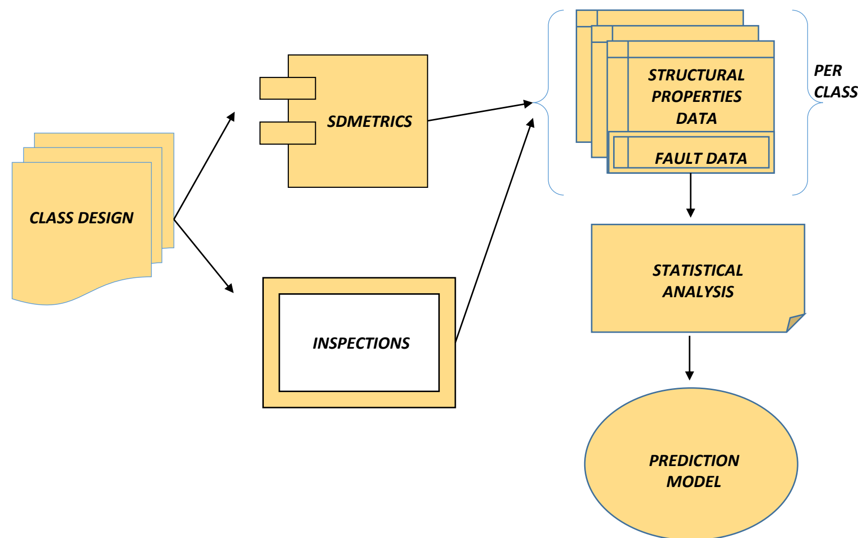

Figure 11: Building a Prediction Model SmellGuru

4.3.9 Operation

All tests are run the *Jupyter Notebook*[9] environment with the *Python* version 3.7.7 and manipulate the dataset using the *Python Data Analysis Library*[10]. Graphs are generated using the *matplotlibrary*[11]. Two learning algorithms were used to the classification task:Random Forest

[9] https://jupyter.org/
[10] https://pandas.pydata.org/
[11] https://matplotlib.org/

(RF) and Support Vector Machine (SVM). This work used the implementation of these algorithms found in *scikit-learnig*[12] library (version 0.23.1).

As this is a multi-class classification problem, is adopt two training approaches for RF. In the first approach, is used the dataset (Section 4.3.4) without any manipulation. In the second approach, is binarize [13] the labels and use the One-Vs-Rest[14] (OvR) strategy to training the model. Since is consider SVM only for comparison, in this algorithm, not apply binarization.

By training the RF model without the binarize step, we created 960 models through the hyper parameterization method. As the standard 5 times cross validation is used, a total of 4,800 models were trained. The model with the best accuracy reached the value of 96% with the following parameters: maximum depth as 13, number of features as the square root of feature number, the minimum number of samples required to be at a leaf node as 2, and the number of trees in the forest as 375. Figure 12 shows the confusion matrix for this scenario.

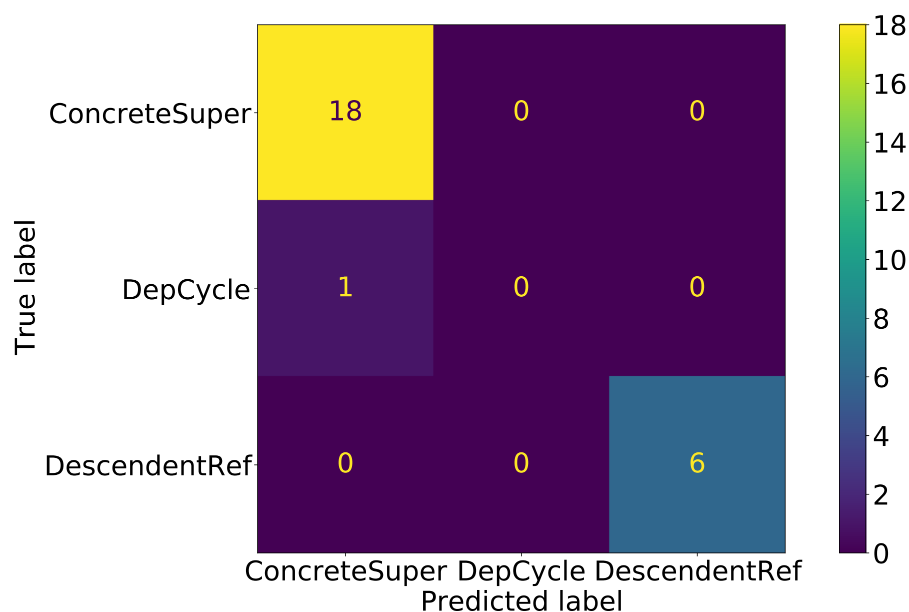

Figure 12: Confusion matrix results of best RF model

However, when trained the model using the standard parameters, without the hyper parameterization method, the accuracy achieved was also 96%. The confusion matrix resulting was the same as show in the Figure 12. When trained the RF model with an OvR strategy, both the accuracy and the confession matrix were the same as the default model, with 96% of accuracy and the same confusion matrix that Figure 12 shows.

By using the RF algorithm, you have the possibility to identify which features are most relevant to the classification task. Figure 15 shows the weight of each characteristic for the model, with the maximum possible value and the sum of the importance of all characteristics equal to 1. As can be seen, several features that are present in the dataset have zero relevance,

[12]https://scikit-learn.org/stable/
[13]https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.label_binarize.html
[14]https://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsRestClassifier.html

and do not appear in the bar graph. The X-axis represents the metrics (features) used to collect design data. The Y-axis represents the importance of the metric to the result. The metrics from Dep_Out and Dep_In were the ones that most contributed to the results.

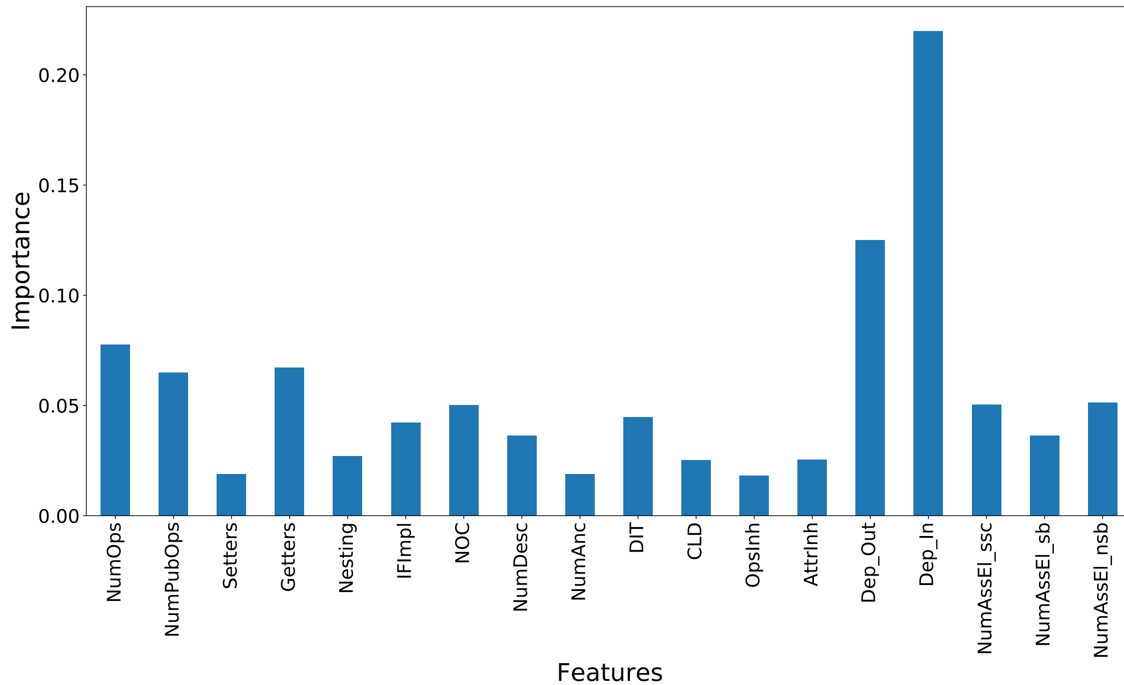

Figure 13: Feature importance to the RF algorithm

As the confusion matrix generated was the same in the tested scenarios, is started to analyze other metrics. Table 13 presents the metrics of f1-score, precision, and recall for both micro and macro averages. The difference in values between micro and macro average may indicate imbalance data, with one or a few classes significantly more present when compared to the total of existing classes in the dataset.

Table 13: Metrics of the RF model with the highest accuracy

Macro			|	**Micro**		
Precision	Recall	F1 - Score	|	Precision	Recall	F1 - Score
0.32	0.33	0.32	|	0.96	0.96	0.96

o finalize the tests is compared the results obtained with the RF using the SVM algorithm. We carry out the training of models with the standard values, with that we have a comparison with the value obtained by the standard model of the RF. We use two of the kernels provided by the implementation of SVM in the library *sklearn*, the Radial Basis Function (RBF) and the Linear kernel. In the first test with the RBF kernel for the classification task, we reached an accuracy of 72%. In the test using the linear kernel, we reached an accuracy of 88%. The confusion matrix resulting from each SVM training is shown in Figure 14. In all tests, the database was split in two, with 80% for training and 20% for the test.

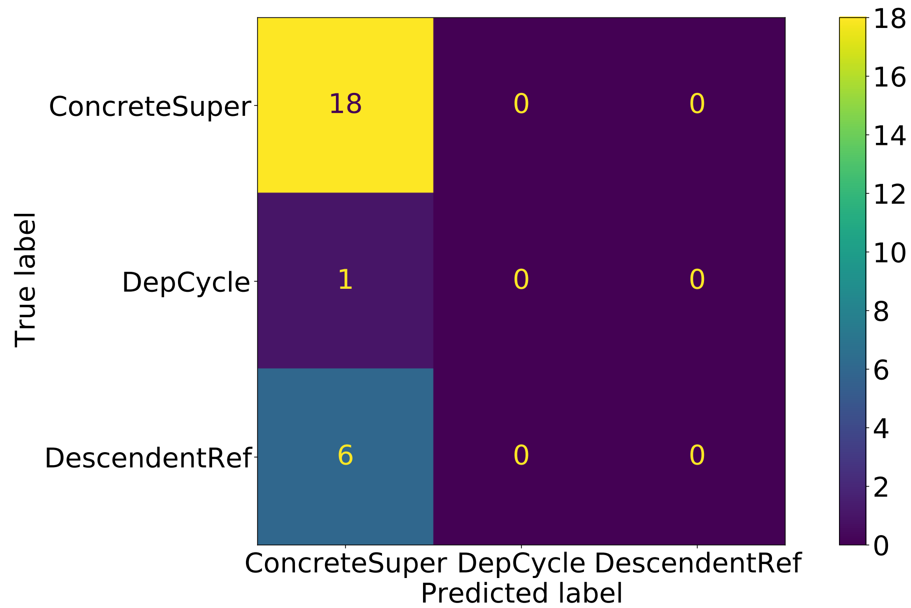

Figure 14: Confusion Matrix - SVM with RBF Kernel

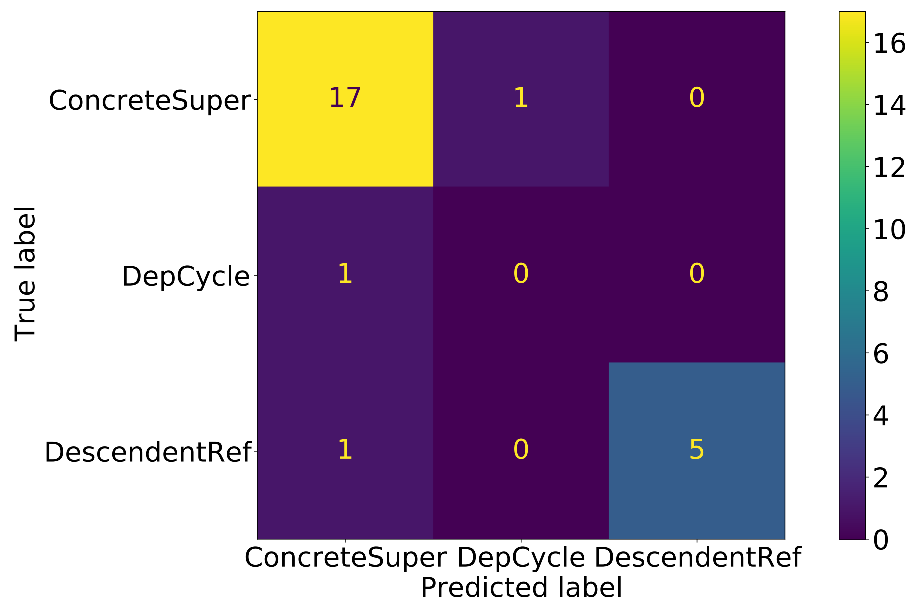

Figure 15: Confusion Matrix - SVM with Linear Kernel

Table 14 presents all results of accuracy obtained in this work. The results obtained by the RF achieve a higher accuracy when compared to the SVM. However, according to the metrics presented in the Table 13, more tests are needed with a larger and more heterogeneous dataset.

Table 14: Accuracy of trained models

RF		**SVM**	
RF	RF OvR	RBG	Linear
96%	96%	72%	88%

5 EVALUATION

This chapter details the evaluation carried out during the development of the proposed SmellGuru approach. To this end, the text will sequentially address the following topics: (Section 5.2) Presentation and discussion, in a previous phase, of the results obtained through the implementation and simulation through proposed scenarios through a DSL for specifying bad smells; (Section 5.3) Presentation and discussion of experimental data related to the design problem prediction model.

5.1 Evaluation SmellDSL

This Section evaluates the implementation of **SmellDSL**..

5.1.1 Research Objective and Questions SmellDSL

This Section evaluates the effects of **SmellDSL** on three variables: the correct rate of the specified smells, the error rate in the specification, and the time required to specify *smells*. These effects are investigated from concrete smell specification scenarios so that empirical results can be generated. With that in mind, the purpose of this Section is established based on the GQM (WOHLIN et al., 2012) model as follows:

To analyze a *SmellDSL* **for the purpose of investigating** its effects **with respect to** correctness rate, error rate and time **from the point of view of** developers **in the context of** smells specification elaboration.

In particular, this Section aims to make an initial analysis of the impact on correctness rate, error rate, and time invested by developers when using *SmellDSL* to specify *bad smells.* In this sense, three research questions (RQ) were formulated:

- **RQ1:** What is the correctness rate in using *SmellDSL* to define *bad smells*?
- **RQ2:** What is the error rate when using *SmellDSL* to define *bad smells*?
- **RQ3:** How long does it take to define *bad smells* using *SmellDSL*?

5.1.2 Study Variables

Correctness Rate (CT). Study participants performed eight experimental tasks. The correctness rate represents the average of correct answers for an experimental task, assuming values from 0 (zero) to 1 (one), where zero represents that no one answered the question correctly and 1 indicates that all participants answered the question correctly. This variable is the average rate of experimental tasks performed correctly.

Error Rate (ET). This metric seeks to measure the error rate when defining a *bad smell*. As well as the correctness rate, it assumes values from 0 (zero) to 1 (one), where zero represents that no one got the question wrong and 1 indicates that all the participants got the question wrong. In scenario 1 (Table 15), each participant needs to create a *Smelltype Bloater*, a *Smell LargeClass* that contains a *feature*. If a participant misses the Smelltype definition, then the error rate for the question will be 1/3 (number of errors divided by the number of elements in the activity).

Time (T). This metric seeks to measure the time required to specify bad smells using *SmellDSL* for each proposed scenario (described in Table 15). The study quantifies the average time taken to complete each question.

5.1.3 Hypotheses and Analysis Procedure

Three hypotheses were formulated from the research questions to allow an initial evaluation of *SmellDSL*. It is conjectured that participants using *SmellDSL* will be able to obtain a correctness rate per experimental task of equal to or greater than 0.5 (50%), error rate greater than or equal to 0.5 (50%), and a maximum time of 15 minutes during the realization of the proposed scenarios. Thus, this initial assessment compares the results obtained with such hypothesized values. To test the hypotheses, the *one sample t-test* method was used. The *Winks* and *IBM SPSS Statistics* tools were used to calculate the *t-test*. The formulated hypotheses helped us to make decisions about *SmellDSL* based on the collection of information from each participant, it is important to note that the results may vary according to the samples obtained as it is still an experiment in its initial phase.

5.1.4 Experimental Tasks

Table 15 presents the evaluation scenarios or experimental tasks defined to evaluate the *SmellDSL*. In total, eight scenarios were specified. Each participant performed each task, totaling 96 cases of evaluation of the use of *SmellDSL*. For example, Scenario 01 contains a task in which the participant must specify a *Smelltype Bloater* and a *Smell LargeClass* containing a *feature*.

5.1.5 Context and Selection of Participants

The evaluation was conducted with 12 participants, four students, and eight professionals from Brazilian companies with professional experience in software development. An email was sent to undergraduate and graduate students at the University of Vale do Rio dos Sinos and the Federal Institute of Mato Grosso (IFMT), selecting those who have experience with software development and modeling. Some participants had a master's and bachelor's degree

Table 15: Evaluation scenarios of *SmellDSL*

	SmellType			Smell									
Scenarios	**Bloaters**	**Dispensables**	**Couplers**	**LargeClass**	**LongParameterList**	**DuplicateCode**	**LongMethod**	**FeatureEnvy**	**Features**	**Symptom**	**Treatment**	**Rule**	**Maintence**
C01	●	○	○	●	○	○	○	○	●	○	○	○	○
C02	●	○	○	○	●	○	●	○	●	○	○	○	○
C03	○	●	○	○	○	●	○	○	●	●	●	●	○
C04	○	●	○	○	○	○	○	●	●	●	●	●	○
C05	○	○	●	○	○	○	○	●	●	●	●	●	○
C06	○	○	●	○	○	○	○	●	●	●	●	●	●
C07	●	●	○	●	●	○	○	●	●	●	●	●	○
C08	●	●	○	●	●	○	○	●	●	●	●	●	●

Legend: (●) *Support* (○) *Not Support*

(or equivalent), as well as knowledge in software modeling and programming. The selected participants, including students, have different profiles and levels of expertise. The experiment was carried out similarly to a practical laboratory exercise. Each participant received the same training on the proposed technique and the experimental procedures to be performed. The profile of each participant is represented in Table 16.

5.1.6 Results SmellDSL

Table 17 presents the initial results for the formulated hypotheses. The bold value of *p-value* indicates that the value was less than 0.05. Figure 16 shows the average time invested per scenario.

Hypothesis 1 (QP1): The result of the *t-test* shows that the mean correctness rate [Mean = 0.6875, SD = 0.16517] was statistically significant at a level of 0.05 of significance (t = 3.21, df = 7, p = 0.015) for the test value equal to 0.5, with a mean difference of 0.015 and 95% Confidence Interval (0.54939, 0.82561) . The null hypothesis that suggested a correctness rate lower than 0.5 can be rejected.

Hypothesis 2 (QP2): The result of the *t-test* also shows that the mean error rate [Mean = 0.3125, SD = 0.16517] was statistically significant at a level of 0, 05 of significance (t = -3.21, df = 7, p = 0.015) for the test value equal to 0.5, with a mean difference of 0.015 and 95% Confidence Interval (0.17439, 0, 45061)]. The null hypothesis that suggested an error rate greater than 0.5 can be rejected.

Hypothesis 3 (QP3): The result of the *t-test* also shows that the time mean [Mean = 9.08333, SD = 2.51346] was statistically significant at a 0.05 significance level (t = -6.66, df = 7, p < 0.001) for the test value equal to 15, with a mean difference of 0.001 and 95% Confidence Interval (0.17439, 0.45061). The null hypothesis that suggested a specification time equal to or greater than 15 minutes can be rejected.

Table 16: Participants profile

Features	**Description**	**%**
Age	from 18 to 25 years	30%
	from 26 to 35 years	10%
	from 36 to 45 years	52%
	More than 45 years	8%
Academic education	Eng. of Computing	10%
	Information System	30%
	Systems Analysis	40%
	Others	20%
Education	Technical	10%
	Graduation	40%
	Master	10%
	Others	40%
Current Position	Programmer	40%
	Analyst	30%
	Manager	20%
	Others	10%
Time in office	less than 2 years	30%
	from 2 to 4 years	20%
	from 5 to 6 years	10%
	more than 8 years	40%
Experience Time Software modeling	less than 2 years	50%
	2-4 years	20%
	from 7 to 8 years	10%
	over 8 years	20%
Software development	less than 2 years	50%
	from 2 to 4 years	10%
	from 7 to 8 years	10%
	over 8 years	30%

Table 17: Initial results for the tested hypotheses.

Variable	Value	N	Mean	Deviation	t	Degree of Freedom(DF)	p-value
H1: Correction	0.5	8	0.6875	0.16517	3.21	7	**0.015**
H2: Error	0.5	8	0.3125	0.16517	-3.21	7	**0.015**
H3: Time	15	8	9.08333	2.51346	-6.66	7	**0.001**

5.1.7 Conclusion and Future Works

This Section introduced *SmellDSL*, a domain-specific language for specifying *bad smells*, along with a supporting tool, the *SmellDSL-tool*. An exploratory study was carried out to understand the impact of *SmellDSL* on three variables, the correctness rate of the specifications

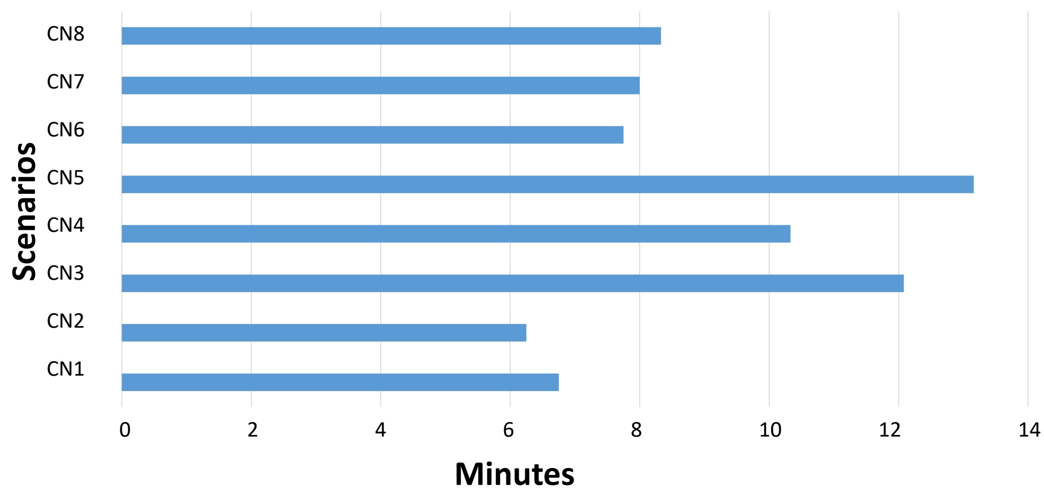

Figure 16: Average time invested per scenario in minutes (RQ3)

created, the error rate and the time invested. The initial results obtained, supported by statistical tests, point to encouraging results when revealing a correctness rate above 50%, an error rate below 30%, and an effort less than 15 minutes to specify a *bad smell*. *SmellDSL* showed promise, but more studies need to be carried out to better evaluate the benefits of the language.

5.2 Evaluation of Model SmellGuru for Predicting Design Problems

This section presents the results obtained after sorting the dataset according to design metrics represented in Section 4.3.4) to answer the formulated research questions in Section 4.3.1).

5.2.1 RQ1: Can the presence of code smells impact the design?

After training the ML model, the Random Forest (RF) algorithm identifies which features are most important related to design smells. This information is collected from smells features with higher incidences correlated to RQ1, the results can be shown in Figure 13, the analyzed data, as well as the metrics, refer to the content present in Table 12.

The number of dependencies where the class is the client and the class is the supplier is featured with greater importance. The catalog of recurring problems can be used as a specific part to identify a possible smell according to the code quality level. Finally, it is important to note that the proposed approach can be applied in virtually any type of software lifecycle. For example, in an agile development environment, the proposed assessment of the practices can be applied at the end of each sprint to identify possible smells.

Cyclically-dependent - This smell arises when two or more abstractions depend on each other directly or indirectly creating a tight coupling between the abstractions. When a set of abstractions are coupled together in a tangle, a change in one of these abstractions may lead to a ripple effect across all the coupled classes. Hence, it is difficult to understand as well as introduce new features or changes to the classes belonging to the tangle. This smell can be refactored by breaking the cycle by moving some of the fields or methods to another class (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014).

5.2.2 RQ2: What is the performance of ML algorithms for predicting impact and non-impact design changes?

When using the RF algorithm, it is a have the possibility to identify which features are most relevant to the classification task. Figure 15 shows the weight of each characteristic for the model, with the maximum possible value and the sum of the importance of all characteristics equal to 1. As can be seen, several features that are present in the dataset have zero relevance and do not appear in the bar graph. The X-axis represents the metrics (features) used to collect design data. The Y-axis represents the importance of the metric to the result. The metrics from Dep_Out and Dep_In were the ones that most contributed to the results.

5.2.3 RQ3: What features are the best indicators of change that impact design?

The features importance which has been obtained by the training samples using the RF algorithm is illustrated in Figure 15. This figure displayed the result for each feature when all features are used as input in the RF. In this study, the feature importance is determined by the mean decrease permutation accuracy. The result shows that The number of dependencies where the de class is the client (Dep_Out) and the number of dependencies where the class is the supplier (Dep_In) sizes appear to be the most relevant features. The result also indicates that the number of operations in a class (NumOps), plays a significant factor in determining the presence of smells. This finding is aligned with results that have been reported by (BRIAND; WÜST, 2002; AL DALLAL, 2017). Therefore, coupling metrics greatly help to identify small parts of a design that contain a large number of faults.

5.2.4 Discussion

This section discusses findings with regard to the projects analyzed. Furthermore, models are built and evaluated using community-aware metrics based on design smells.With the analysis made by this work, it was possible to find evidence that the occurrences of known code smells had a positive impact on the identification of design smells. It seems worthwhile to investigate different dimensions of coupling: import and export coupling. In general, the possible

implications can be related to the standards established in the project and in the process. Some of these standards and their applications are detailed in the next paragraphs. There are few works that analyze the relationships between code smells and their implications for software design. This behavior, discovered in this empirical study, can be used to optimize the detection of design smells. In general, we can use the possible smells, and through traceability in the use of these objects with possible code anomalies, we can help developers in future software fixes, prioritizing their demands related to smells.

Another possible practical implication concerns the planning of refactoring activities. From the discussion, it is observed that most of the smells occur in the coupling category in entities affected by smells. This identification can aid in the strategy and prioritization of software fixes and maintenance. This implies that the removal of these smells can impact the use of software refactoring techniques, established in the project.

5.2.4.1 Conclusions

Design smells arise from poor design decisions that make the design fragile and difficult to maintain. Classification or quantification of smells that impact design is important to help in pointing out deeper systemic problems that lead to negative outcomes. Therefore, is presented a study on using machine learning to identify design impact. Predicting, and classifying design smells types using RF algorithm. The algorithm has been tested on secondary data that contained several features. Approximately 10.000 datasets have been used in this study and the performance of the proposed method has been measured. The obtained accuracy of the proposed method was 96% (RF). It can be concluded that RF can accurately classify smells, however further research is required to analyze and determine the impact on design. Furthermore, for future work, this study attempts to compare various types of methods in determining the accuracy and effectiveness of the RF algorithm as well as verify the results that have been obtained.

5.3 Evaluation SmellGuru Proposed Model

This section highlights the results of the approach SmellGuru evaluation. For this, it is applied a questionnaire that aims to investigate the usefulness of the SmellGuru approach for predicting Bad Smells. In this sense, the answers to the questions elaborated were based on the experience of the 23 participants, acquired during the presentation of the SmellGuru approach. The questionnaire has two parts: (1) The first search characterizes the participant and can be represented in Table 18; and (2) The second part seeks to collect information about the perception of the practical usefulness of SmellGuru and is represented in Table 19. Both parties are anonymous.

5.3.1 Context and Selection of Participants

The evaluation was conducted with 23 participants, students, and professionals from Brazilian companies with professional experience in software development. An email was sent to undergraduate and graduate students at the University of Vale do Rio dos Sinos and the Federal Institute of Mato Grosso (IFMT), selecting those who have experience with software development and modeling. Some participants had a master's and bachelor's degree (or equivalent), as well as knowledge in software modeling and programming. We chose participants, including students, so that we could have different profiles and levels of expertise. The questionnaire was carried out in a similar way to a practical laboratory exercise. Each participant received the same training on the proposed SmellGuru approach and the experimental procedures to be performed. The profile of each participant is represented in Table 18.

Table 18 presents the results collected after the application of the TAM questionnaire. Participant profile. After the application of the experimental and feedback collection phases, in a group of twenty-three participants, it was possible to verify that half was composed of individuals aged between 26 and 35 years old, while the second half consisted of aged between 36 and 45 years. Mainly with training in Information Systems (30.4%), followed by Systems Analyst (21.7%) and Computer Science (8.7% each). Regarding professional experience, the majority (47.8%) had more than 2 years of experience in software development. When asked about their perception that if the identification of bad smells had more formalism in the software design, developers would start to use it more frequently, there was (56.52%) agreement.

Emphasize the results described in Table 19 the information related to the use of software for the identification of bad smells in the companies where the participants work. This information characterizes the experience with the identification of smells by each participant as described below:

- I use software to detect bad smells (R1);
- Bad smell identification is not yet universally accepted as a standard in software development projects (R2);
- Bad smell identification models help during discussions about how to maintain applications (R3);
- Is the identification of bad smells growing? (R4)
- There is no "role" (e.g., programmer, analyst, etc) that particularly requires standards for identifying bad smells (R5);
- The cost of promoting the correct understanding for the identification of bad smells among different people with different levels of training/experience and ways of thinking is high in software design (R6);

Table 18: Participants profile

Features	Description	%
Age	from 18 to 25 years	8,7%
	from 26 to 35 years	47.8%
	from 36 to 45 years	26.1%
	More than 45 years	17.4%
Academic education	Eng. of Computing	8.7%
	Information System	30.4%
	Systems Analysis	21.7%
	Others	39.1%
Education	Technical	8.7%
	Graduation	30.4%
	Specialization	39.1%
	Master's	17.4%
	Others	4.3%
Current Position	Programmer	17.4%
	Analyst	26.1%
	Others	56.5%
Time in office	less than 2 years	13%
	from 2 to 4 years	26.1%
	from 5 to 6 years	8.7%
	from 7 to 8 years	13%
	more than 8 years	39.1%
Experience Time Software modeling	less than 2 years	43.5%
	2-4 years	8.7%
	from 5 to 6 years	8.7%
	from 7 to 8 years	13%
	over 8 years	26.1%

- Bad smells identification processes tend to be used either in the design phase or in the implementation phase, never in both (R7);
- Those who use the different ways of identifying bad smells during the design phase tend to use it selectively and informally (R8);
- The developer tends to use the identification of bad smells to express abstractions and the most critical cases. But they stop using it when they start thinking more concretely, especially when they're implementing (R9);
- Bad smells identification tools are not usual (R10);
- If the identification of bad smells had more formalism in the software project, developers would use it more frequently (R11);
- The identification of bad smells is not necessary for the needs that software development

projects present (R12);

Table 19: Participants Experience

Description	I agree	Part. agree	Neutral	Part. Disagree	Strongly Disagree
Regarding the use of software to identify bad smells in companies:					
R1.	43.47%	13.04%	21.73%	8.69%	13.04%
R2.	34.78%	39.13%	21.73%	4.34%	0
R3.	43.47%	39.13%	13.04%	0	4.34%
R4.	34.78%	39.13%	21.73%	4.34%	0
R5.	30.43%	21.73%	26.08%	21.73%	0
R6.	52.17%	13.04%	8.69%	21.73%	4.34%
R7.	34.78%	30.43%	8.69%	8.69%	17.39%
R8.	34.78%	21.73%	30.43%	4.34%	8.69%
R9.	34.78%	17.39%	39.13%	4.34%	4.34%
R10.	17.39%	39.13%	21.73%	13.04%	8.69%
R11.	56.52%	4.34%	21.73%	13.04%	4.34%
R12.	13.04%	21.73%	21.73%	26.08%	17.39%

Through the application of the TAM, it was possible to evaluate the perceived ease of use, perceived usefulness, and behavioral intent of using the proposed SmellGuru approach. As shown in Table 20, respondents agree that SmellGuru is easy to interpret (43.47%) and innovative (60.86%) . SmellGuru would make it easier to maintain software (78.26%). There was unanimously in pointing out that the purpose of SmellGuru was clear to me.

Table 20 presents the results related to the acceptance of the SmellGuru technology, for that, questionnaires were prepared for the participants regarding the 03 (tree) items described below:

- **Perceived ease of use (Q1)**
  - I found SmellGuru easy to interpret (Q1.1);
  - I found SmellGuru easy to integrate with other technologies (Q1.2.);
  - I found SmellGuru innovative (Q1.3)
- **usefulness perception (Q2)**
  - SmellGuru would make software maintenance easier (Q2.1);
  - SmellGuru would help with productivity (Q2.2)
  - SmellGuru would reduce code anomaly identification time (Q2.3);
- **Behavior Intent (Q3)**

- I would use the SmellGuru approach as a support tool in automatic software maintenance (Q3.1);

- **Clarity of the Proposed Approach (Q4)**
  - SmellGuru's purpose became clear to me (Q4.1);
  - The SmellGuru steps were properly understood (Q4.2);

Table 20: TAM (Technology Acceptance)

Description	I agree	Part. agree	Neutral	Part. Disagree	Strongly Disagree
Q1. Perceived ease of use					
Q1.1.	43.47%	47.82%	8.69%	0	0
Q1.2.	30.43%	47.82%	21.73%	0	0
Q1.3.	60.86%	21.73%	13.04%	0	4.34%
Q2. Usefulness perception					
Q2.1.	78.26%	13.04%	8.69%	0	0
Q2.2.	65.21%	26.08	8.69%	0	0
Q2.3.	73.91%	17.39%	8.69%	0	0
Q3. Behavior intention					
Q3.1.	65.21%	30.43%	0	0	4.34%
Q4. Clarity of the approach					
Q4.1.	95.65%	0	0	0	4.35%
Q4.2.	95.65%	0	0	0	4.35%

According to the results of the information contained in Table 21, two important particularities are highlighted about the proposed approach by SmellGuru: (1) SmellGuru steps have been properly understood, and (2) clarity of the proposed approach. For this, participants who have different professional specialties, experiences, and training help to answer the problem (P2) lack of an overview of approaches to predict design problems, as seen in the Introduction of this work.

Table 21: TAM (Clarity)

Description	Percentage
Steps have been properly understood	95.65%
Clarity of the proposed approach	95.65%

6 CONCLUSION AND FUTURE WORK

Different types of software design problems were analyzed and identified according to the bibliographic references, there are many, each with its own motivation and different characteristics. Thus causing more and more types of code problems to arise according to the individual differences of each software project. We verified that the libraries used for model validation can be one of the available options of computational tools, ideal for us to identify the types of anomalies, according to the artifact information of each software project. When implementing this work, is shown that the use of intelligent machines is a topic that is increasingly studied and implemented in everyday reality, ceasing to be something purely academic. There is the use of artificial intelligence in turnstile software for shopping malls, stock exchanges, aviation, etc.

And for that, currently, in the identification of source code anomalies, the so-called static code analysis tools are used. These carry information like the human expert, and in the decision-making process, all psychologism is eliminated. Because it does not carry human subjectivity and sentimentality, it does not allow for a better interpretation of reality, in particular, the metric tools used to identify smells, our implemented model allows a better judgment about the types of code smells, as we are not locked into a single problem identification database. Demonstrating in this model that it was possible to carry out tests, training, and evaluation in order to have a good statistic about its efficiency.

During the development of this work, it is essential to understand the concept of bad smells and design smells. As highlighted in Chapter 1, the definition of smells presented for Software Engineering due to the dynamics related to software production. In addition, we implemented a tool to automate problem definition, as we consider that the design tool seeks a manual of the source code of these problems is cumbersome, complex, and prone to failure. The tool was validated with the elaboration of experimental scenarios. SmellDSL became efficient according to the resources for defining and understanding bad smells according to their need to define and adapt the project to be developed.

However, when executing the prototype and modeling the proposed model scenarios, it is shown that it can cover any necessary needs related to possible behavioral changes arising from the new bad smells. When analyzing software projects using the model trained to identify bad smells, potential benefits from defined and already cataloged features related to Bad Smells, detected occurrences, for example, reduced coupling, increased cohesion, greater code reuse, encapsulation of features, and reduction of code complexity, can influence the occurrence of design smells.

Some future works are proposed aiming at the evolution of the proposed approach and support in the prediction of design problems. Therefore, the following works are suggested:

- DSL **integration** with the machine learning model.
- Evaluate the **DSL** implemented with other bad smells specification scenarios.

- To analyze the **bad smells relationships** and their implications for software design.

6.1 Contribuitions

The main contribution brought by this dissertation is the presentation of a approach to help identify and predict design problems using machine learning techniques. Implementation of a DSL to specify bad smells. In addition, the specific contributions brought by the approach are:

Intelligible workflow to assist those involved in software design, providing clear guidance and facilitating the inclusion of new strategies for predicting design problems.

DSL implementation for specifying bad smells was designed with some assumptions in mind: (1) support for the specification of bad smells already cataloged (in an informal way) in the literature, enhancing a broader characterization; (2) use of notation and concepts found in object-oriented languages, facilitating their use by developers already familiar with object-oriented languages; and (3) separate the definition from the characteristics of bad smells and the rules to identify them, allowing bad smells to be identified differently by different development teams.

Design problem prediction model was implemented using machine learning techniques. It is important to note that the steps at this time are not integrated with DSL. Unfortunately, bad smells are typically defined and specified informally, making it difficult to understand and refactor tasks. The focus is only on supervised learning.

6.2 Limitations

This study investigated in its course questions essentially about the detection and prediction of Design Smells problems. Although the contributions of this study are noticeable, its approach needs improvement, since the development time of this study is not enough to cover all the existing gaps. Thus, the main limitations identified in this study are presented, as well as suggestions for future work in order to cover these points.

Experimental limitations. There is a need to expand the number of participants as well as increase the participation of professionals linked to the industrial software sector in the surveys. The scenarios presented have a reduced number of elements, that is, they are small models that for the most part do not match the current situation in the industrial scope for the identification and definition of bad smells. Therefore, in future work, there is a need to implement richer models, which present the largest number of elements and characteristics for the definition of bad smells.

Limitations of the prototype. However, the participants used a prototype and carried out the implementation and definition of bad smells. The SmellDSL tool is 100% functional as an Eclipse platform plugin. The three main limitations: (1) it allows specifying bad smells, but does not generate code to automate the quantification of bad smells; (2) integrated only with

the Eclipse platform, and can be supported in other IDEs, such as VS code, Spring Tool Suite, among others; (3) it has a well-defined grammar using BNF, but language documentation and creating examples for developers are limitations;

Limitations of the technique. Although the SmellGuru approach is semi-automatic for the integration of a prediction model of design problems, being effective about the manual technique, there is a need to expand and deepen the phase of studies, this is because decisions suffer human interference and are subject to error, as demonstrated in the experiment, that is, some of the models for training may have been produced incorrectly. Thus, in future work, it is observed the need for new experiments in view of the execution of the automatic SmellGuru approach, having emphasis on analyzing its precision and accuracy in relation to the proposal of anticipating Design Smells, and finally coupling other techniques to the prototype, such as the use of artificial intelligence to define variations of Bad Smells, given the challenge of making better and more accurate decisions about the feature models needed to define the occurrence of Bad Smells is still not well defined by software development teams.

REFERENCES

ABBES, M. et al. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: EUROPEAN CONF. ON SOFTWARE MAINTENANCE AND REENGINEERING, 15., 2011. **Anais. . .** . p. 181–190.

AL DALLAL, J. Predicting move method refactoring opportunities in object-oriented code. **Information and Software Technology**, v. 92, p. 105–120, 2017.

AL-QUDAH, S.; MERIDJI, K.; AL-SARAYREH, K. T. A comprehensive survey of software development cost estimation studies. In: OF THE INTERNATIONAL CONFERENCE ON INTELLIGENT INFORMATION PROCESSING, SECURITY AND ADVANCED COMMUNICATION, 2015. **Proceedings. . .** . p. 1–5.

AL-SHAABY, A.; ALJAMAAN, H.; ALSHAYEB, M. Bad smell detection using machine learning techniques: a systematic literature review. **Arabian Journal for Science and Engineering**, v. 45, n. 4, p. 2341–2369, 2020.

ALENEZI, M. et al. **Test suite effectiveness**: an indicator for open source software quality. 2016. 1–5 p.

ALJEDAANI, W. et al. Test smell detection tools: a systematic mapping study. **Evaluation and Assessment in Software Engineering**, p. 170–180, 2021.

ALKHARABSHEH, K. et al. **Improving design smell detection for adoption in industry**. 2018. 213–218 p.

ALKHARABSHEH, K. et al. Software design smell detection: a systematic mapping study. **Software Quality Journal**, v. 27, n. 3, p. 1069–1148, 2019.

ANSARI, F.; GLAWAR, R.; SIHN, W. Prescriptive maintenance of cpps by integrating multimodal data with dynamic bayesian networks. **Machine Learning for Cyber Physical Systems, Technologies for Intelligent Automation**, v. 11, p. 1–8, 2020.

AZEEM, M. I. et al. Machine learning techniques for code smell detection: a systematic literature review and meta-analysis. **Information and Software Technology**, v. 108, p. 115–138, 2019.

BARBOSA, C. et al. Revealing the social aspects of design decay: a retrospective study of pull requests. In: ______. **Proceedings of the 34th brazilian symposium on software engineering**. p. 364–373.

BARRIGA, A. et al. Addressing the trade off between smells and quality when refactoring class diagrams. **J. Object Technol**, v. 20, n. 3, p. 1, 2021.

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software architecture in practice**.

BETTINI, L. et al. Supporting safe metamodel evolution with edelta. **Int. Journal on Software Tools for Technology Transfer**, v. 24, n. 2, p. 247–260, 2022.

BOEHM, B.; ROSENBERG, D.; SIEGEL, N. **Critical quality factors for rapid, scalable, agile development**. 2019. 514–515 p.

BOUAZIZ, A. et al. Short text classification using semantic random forest. In: INTERNATIONAL CONFERENCE ON DATA WAREHOUSING AND KNOWLEDGE DISCOVERY, 2014. **Anais...**. p. 288–299.

BOUSDEKIS, A. et al. A unified architecture for proactive maintenance in manufacturing enterprises. In: **Enterprise interoperability viii**. p. 307–317.

BOUSSAA, M. et al. Competitive coevolutionary code-smells detection. In: INT. SYMPOSIUM ON SEARCH BASED SOFTWARE ENGINEERING, 2013. **Anais...**. p. 50–65.

BREIMAN, L. Random forests. **Machine learning**, v. 45, n. 1, p. 5–32, 2001.

BRIAND, L. C.; WÜST, J. Empirical studies of quality models in object-oriented systems. **Advances in computers**, v. 56, p. 97–166, 2002.

CACHADA, A. et al. Maintenance 4.0: intelligent and predictive maintenance system architecture. In: IEEE 23RD INTERNATIONAL CONFERENCE ON EMERGING TECHNOLOGIES AND FACTORY AUTOMATION (ETFA), 2018., 2018. **Anais...**. v. 1, p. 139–146.

CARAM, F. L. et al. Machine learning techniques for code smells detection: a systematic mapping study. **International Journal of Software Engineering and Knowledge Engineering**, v. 29, n. 02, p. 285–316, 2019.

CHEN, C. et al. **How do defects hurt qualities? an empirical study on characterizing a software maintainability ontology in open source software**. 2018. 226–237 p.

CHOUDRIE, J. et al. Machine learning techniques and older adults processing of online information and misinformation: a covid 19 study. **Computers in Human Behavior**, v. 119, p. 106716, 2021.

DALZOCHIO, J. et al. Machine learning and reasoning for predictive maintenance in industry 4.0: current status and challenges. **Computers in Industry**, v. 123, p. 103298, 2020.

DAS, A. K.; YADAV, S.; DHAL, S. Detecting code smells using deep learning. In: TENCON 2019-2019 IEEE REGION 10 CONF. (TENCON), 2019. **Anais...**. p. 2081–2086.

DEVI, D. G.; PUNITHAVALLI, M. **A hierarchical method for detecting codeclone**. 2011. 126–128 p. v. 1.

DI NUCCI, D. et al. Detecting code smells using machine learning techniques: are we there yet? In: 2018 IEEE 25TH INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION AND REENGINEERING (SANER), 2018. **Anais...**. p. 612–621.

DINAN, E. et al. Anticipating safety issues in e2e conversational ai: framework and tooling. **arXiv preprint arXiv:2107.03451**, 2021.

EL KOUTBI, S.; IDRI, A.; ABRAN, A. Systematic mapping study of dealing with error in software development effort estimation. In: EUROMICRO CONFERENCE ON SOFTWARE ENGINEERING AND ADVANCED APPLICATIONS (SEAA), 2016., 2016. **Anais...**. p. 140–147.

EMDEN, E. van; MOONEN, L. Java quality assurance by detecting code smells. In: NINTH WORKING CONFERENCE ON REVERSE ENGINEERING, 2002. PROCEEDINGS., 2002. **Anais...**. p. 97–106.

ERTURK, E.; SEZER, E. A. Iterative software fault prediction with a hybrid approach. **Applied Soft Computing**, v. 49, p. 1020–1033, 2016.

FARD, A. M.; MESBAH, A. Jsnose: detecting javascript code smells. In: IEEE 13TH INTERNATIONAL WORKING CONFERENCE ON SOURCE CODE ANALYSIS AND MANIPULATION (SCAM), 2013., 2013. **Anais...**. p. 116–125.

FENTON, N.; NEIL, M. A critique of software defect prediction models. **IEEE Transactions on Software Engineering**, v. 25, n. 5, p. 675–689, 1999.

FERNANDES, E. et al. A review-based comparative study of bad smell detection tools. , 2010.

FERNANDES, E. et al. A review-based comparative study of bad smell detection tools. In: INTERNATIONAL CONFERENCE ON EVALUATION AND ASSESSMENT IN SOFTWARE ENGINEERING, 20., 2016. **Proceedings...**. p. 1–12.

FONTANA, F. A.; BRAIONE, P.; ZANONI, M. Automatic detection of bad smells in code: an experimental assessment. **J. Object Technol.**, v. 11, n. 2, p. 5–1, 2012.

FONTANA, F. A. et al. Arcan: a tool for architectural smells detection. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ARCHITECTURE WORKSHOPS (ICSAW), 2017., 2017. **Anais...**. p. 282–285.

FONTANA, F. A.; ZANONI, M. Code smell severity classification using machine learning techniques. **Knowledge-Based Systems**, v. 128, p. 43–58, 2017.

FOWLER, M. **Refactoring**: improving the design of existing code.

FOWLER, M. et al. **Refactoring**: improving the design of existing code, ser.

GARCIA, J. et al. Identifying architectural bad smells. In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, 2009., 2009. **Anais...**. p. 255–258.

GIRAY, G. A software engineering perspective on engineering machine learning systems: state of the art and challenges. **Journal of Systems and Software**, p. 111031, 2021.

GONÇALES, L. J. et al. Comparison of design models: a systematic mapping study. **International Journal of Software Engineering and Knowledge Engineering**, v. 25, n. 09n10, p. 1765–1769, 2015.

GONÇALES, L. J. et al. Comparison of software design models: an extended systematic mapping study. **ACM Computing Surveys**, v. 52, n. 3, p. 1–41, 2019.

GONÇALES, L. J.; FARIAS, K.; SILVA, B. C. da. Measuring the cognitive load of software developers: an extended systematic mapping study. **Information and Software Technology**, v. 136, p. 106563, 2021.

GRIFFITH, I.; WAHL, S.; IZURIETA, C. **Evolution of legacy system comprehensibility through automated refactoring**. 2011. 35–42 p.

HADJ-KACEM, M.; BOUASSIDA, N. **A hybrid approach to detect code smells using deep learning.** 2018. 137–146 p.

HAJI, S. H.; AMEEN, S. Y. Attack and anomaly detection in iot networks using machine learning techniques: a review. **Asian Journal of Research in Computer Science**, p. 30–46, 2021.

HEGEDŰS, C.; VARGA, P.; MOLDOVÁN, I. The mantis architecture for proactive maintenance. In: INTERNATIONAL CONFERENCE ON CONTROL, DECISION AND INFORMATION TECHNOLOGIES (CODIT), 2018., 2018. **Anais. . .** p. 719–724.

HERMANN, M.; PENTEK, T.; OTTO, B. Design principles for industrie 4.0 scenarios. In: HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES (HICSS), 2016., 2016. **Anais. . .** p. 3928–3937.

HUSSAIN, S.; KEUNG, J.; KHAN, A. A. Software design patterns classification and selection using text categorization approach. **Applied soft computing**, v. 58, p. 225–244, 2017.

IBARRA, S.; MUÑOZ, M. Support tool for software quality assurance in software development. In: INT. CONF. ON SOFTWARE PROCESS IMPROVEMENT, 7., 2018. **Anais. . .** p. 13–19.

IRWANTO, D. Visual indicator component software to show component design quality and characteristic. In: SECOND INT. CONF. ON ADVANCES IN COMPUTING, CONTROL, AND TELECOMMUNICATION TECHNOLOGIES, 2010. **Anais. . .** p. 50–54.

JALALI, S.; WOHLIN, C. Systematic literature studies: database searches vs. backward snowballing. In: ACM-IEEE INT. SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT, 2012., 2012. **Proceedings. . .** p. 29–38.

JIARPAKDEE, J. et al. An empirical study of model-agnostic techniques for defect prediction models. **IEEE Transactions on Software Engineering**, p. 1–1, 2020.

KABIR, M. et al. An empirical research on sentiment analysis using machine learning approaches. **International Journal of Computers and Applications**, v. 43, n. 10, p. 1011–1019, 2021.

KANG, J.; RYU, D.; BAIK, J. Predicting just-in-time software defects to reduce post-release quality costs in the maritime industry. **Software: Practice and Experience**, v. 51, n. 4, p. 748–771, 2021.

KAUR, A. et al. A review on machine-learning based code smell detection techniques in object-oriented software system (s). **Recent Advances in Electrical & Electronic Engineering (Formerly Recent Patents on Electrical & Electronic Engineering)**, v. 14, n. 3, p. 290–303, 2021.

KAUR, J. et al. Machine learning techniques for 5g and beyond. **IEEE Access**, v. 9, p. 23472–23488, 2021.

KEELE, S. et al. **Guidelines for performing systematic literature reviews in software engineering**.

KESSENTINI, M.; OUNI, A. **Detecting android smells using multi-objective genetic programming**. 2017. 122–132 p.

KESSENTINI, M.; VAUCHER, S.; SAHRAOUI, H. Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In: IEEE/ACM INT. CONF. ON AUTOMATED SOFTWARE ENGINEERING, 2010. **Proceedings. . .** p. 113–122.

KHOMH, F.; DI PENTA, M.; GUEHENEUC, Y.-G. An exploratory study of the impact of code smells on software change-proneness. In: WORKING CONFERENCE ON REVERSE ENGINEERING, 2009., 2009. **Anais. . .** p. 75–84.

KHOMH, F. et al. Bdtex: a gqm-based bayesian approach for the detection of antipatterns. **Journal of Systems and Software**, v. 84, n. 4, p. 559–572, 2011.

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

KITCHENHAM, B. A. **Systematic review in software engineering**: where we are and where we should be going. 2012. 1–2 p.

KITCHENHAM, B. A.; BUDGEN, D.; BRERETON, O. P. Using mapping studies as the basis for further research–a participant-observer case study. **Information and Software Technology**, v. 53, n. 6, p. 638–651, 2011.

KOKOL, P.; KOKOL, M.; ZAGORANSKI, S. Code smells: a synthetic narrative review. **arXiv preprint arXiv:2103.01088**, 2021.

KOTSIANTIS, S. B.; ZAHARAKIS, I. D.; PINTELAS, P. E. Machine learning: a review of classification and combining techniques. **Artificial Intelligence Review**, v. 26, n. 3, p. 159–190, 2006.

KREIMER, J. Adaptive detection of design flaws. **Electronic Notes in Theoretical Computer Science**, v. 141, n. 4, p. 117–136, 2005.

KRISHNA, R.; MENZIES, T.; LAYMAN, L. Less is more: minimizing code reorganization using xtree. **Information and Software Technology**, v. 88, p. 53–66, 2017.

KUUTILA, M. et al. Time pressure in software engineering: a systematic review. **Information and Software Technology**, v. 121, p. 106257, 2020.

LACERDA, G. et al. Code smells and refactoring: a tertiary systematic review of challenges and observations. **Journal of Systems and Software**, v. 167, p. 110610, 2020.

LENARDUZZI, V. et al. **Are sonarqube rules inducing bugs?** 2020. 501–511 p.

LEW, S.-L. et al. Usability factors predicting continuance of intention to use cloud e-learning application. **Heliyon**, v. 5, n. 6, p. e01788, 2019.

LI, J. et al. Software defect prediction via convolutional neural network. In: INT. CONF. ON SOFTWARE QUALITY, RELIABILITY AND SECURITY, 2017. **Anais. . .** p. 318–328.

LI, Z.; WANG, Y.; WANG, K.-S. Intelligent predictive maintenance for fault diagnosis and prognosis in machine centers: industry 4.0 scenario. **Advances in Manufacturing**, v. 5, n. 4, p. 377–387, 2017.

LIPOW, M. Prediction of software failures. **Journal of Systems and Software**, v. 1, p. 71–75, 1979.

LIU, H. et al. Schedule of bad smell detection and resolution: a new way to save effort. **IEEE transactions on Software Engineering**, v. 38, n. 1, p. 220–235, 2011.

LIU, H. et al. Are smell-based metrics actually useful in effort-aware structural change-proneness prediction? an empirical study. In: ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE (APSEC), 2018., 2018. **Anais...** p. 315–324.

LUJAN, S. et al. A preliminary study on the adequacy of static analysis warnings with respect to code smell prediction. In: ACM SIGSOFT INTERNATIONAL WORKSHOP ON MACHINE-LEARNING TECHNIQUES FOR SOFTWARE-QUALITY EVALUATION, 4., 2020. **Proceedings...** p. 1–6.

MACIA, I. et al. On the relevance of code anomalies for identifying architecture degradation symptoms. In: EUROPEAN CONF. ON SOFTWARE MAINTENANCE AND REENGINEERING, 2012., 2012. **Anais...** p. 277–286.

MACIA, I. et al. **Are automatically-detected code anomalies relevant to architectural modularity? an exploratory analysis of evolving systems**. 2012. 167–178 p.

MANEERAT, N.; MUENCHAISRI, P. **Bad-smell prediction from software design model using machine learning techniques**. 2011. 331–336 p.

MARTÍNEZ-FERNÁNDEZ, S. et al. Continuously assessing and improving software quality with software analytics tools: a case study. **IEEE access**, v. 7, p. 68219–68239, 2019.

MENZEN, J. P.; FARIAS, K.; BISCHOFF, V. Using biometric data in software engineering: a systematic mapping study. **Behaviour & Information Technology**, v. 40, n. 9, p. 880–902, 2021.

MOHA, N. et al. Decor: a method for the specification and detection of code and design smells. **IEEE Transactions on Software Engineering**, v. 36, n. 1, p. 20–36, 2009.

MORENO-INDIAS, I. et al. Statistical and machine learning techniques in human microbiome studies: contemporary challenges and solutions. **Frontiers in Microbiology**, v. 12, p. 277, 2021.

MULLER, S. C.; FRITZ, T. Using (bio) metrics to predict code quality online. In: IEEE/ACM 38TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 2016., 2016. **Anais...** p. 452–463.

MUMTAZ, H. et al. An empirical study to improve software security through the application of code refactoring. **Information and Software Technology**, v. 96, p. 112–125, 2018.

MUMTAZ, H.; SINGH, P.; BLINCOE, K. A systematic mapping study on architectural smells detection. **Journal of Systems and Software**, v. 173, p. 110885, 2021.

NAGAPPAN, N.; BALL, T.; ZELLER, A. Mining metrics to predict component failures. In: INT. CONF. ON SOFTWARE ENGINEERING, 28., 2006. **Proceedings...** p. 452–461.

OIZUMI, W. et al. Code anomalies flock together: exploring code anomaly agglomerations for locating design problems. In: IEEE/ACM 38TH INT. CONF. ON SOFTWARE ENGINEERING (ICSE), 2016., 2016. **Anais...** p. 440–451.

PAIVA, T. et al. On the evaluation of code smells and detection tools. **Journal of Software Engineering Research and Development**, v. 5, n. 1, p. 1–28, 2017.

PALOMBA, F. et al. **Do they really smell bad? a study on developers' perception of bad code smells**. 2014. 101–110 p.

PALOMBA, F. et al. Toward a smell-aware bug prediction model. **IEEE Transactions on Software Engineering**, v. 45, n. 2, p. 194–218, 2017.

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

PAULO SOBRINHO, E. V. de; DE LUCIA, A.; ALMEIDA MAIA, M. de. A systematic literature review on bad smells—5 w's: which, when, what, who, where. **IEEE Transactions on Software Engineering**, 2018.

PECORELLI, F.; Di Nucci, D. Adaptive selection of classifiers for bug prediction: a large-scale empirical analysis of its performances and a benchmark study. **Science of Computer Programming**, v. 205, p. 102611, 2021.

PETERSEN, K. et al. Systematic mapping studies in software engineering. In: INT. CONF. ON EVALUATION AND ASSESSMENT IN SOFTWARE ENGINEERING (EASE) 12, 12., 2008. **Anais. . .** p. 1–10.

PETERSEN, K.; VAKKALANKA, S.; KUZNIARZ, L. Guidelines for conducting systematic mapping studies in software engineering: an update. **Information and Software Technology**, v. 64, p. 1–18, 2015.

PETRE, M. "no shit" or "oh, shit!": responses to observations on the use of uml in professional practice. **Software & Systems Modeling**, v. 13, n. 4, p. 1225–1235, 2014.

PHAN, A. V. et al. Automatically classifying source code using tree-based approaches. **Data & Knowledge Engineering**, v. 114, p. 12–25, 2018.

PLOSCH, R. et al. On the relation between external software quality and static code analysis. In: ANNUAL SOFTWARE ENGINEERING WORKSHOP, 32., 2008. **Anais. . .** p. 169–174.

POPOOLA, S.; ZHAO, X.; GRAY, J. Evolution of bad smells in labview graphical models. **J. Object Technol.**, v. 20, n. 1, p. 1–1, 2021.

PRÄHOFER, H. et al. Static code analysis of iec 61131-3 programs: comprehensive tool support and experiences from large-scale industrial application. **IEEE Transactions on Industrial Informatics**, v. 13, n. 1, p. 37–47, 2016.

RAJKOVIC, K.; ENOIU, E. Nalabs: detecting bad smells in natural language requirements and test specifications. **arXiv preprint arXiv:2202.05641**, 2022.

RASOOL, G.; ARSHAD, Z. A review of code smell mining techniques. **Journal of Software: Evolution and Process**, v. 27, n. 11, p. 867–895, 2015.

RASOOL, G.; ARSHAD, Z. A lightweight approach for detection of code smells. **Arabian Journal for Science and Engineering**, v. 42, n. 2, p. 483–506, 2017.

REIS, J. P. dos; ABREU, F. B. e; CARNEIRO, G. d. F. **Code smells detection 2.0**: crowdsmelling and visualization. 2017. 1–4 p.

RIVAS, A. et al. A predictive maintenance model using recurrent neural networks. In: INTERNATIONAL WORKSHOP ON SOFT COMPUTING MODELS IN INDUSTRIAL AND ENVIRONMENTAL APPLICATIONS, 2019. **Anais...**. p. 261–270.

RUBERT, M.; FARIAS, K. On the effects of continuous delivery on code quality: a case study in industry. **Computer Standards & Interfaces**, v. 81, p. 103588, 2022.

RWEMALIKA, R. et al. Smells in system user interactive tests. **arXiv preprint arXiv:2111.02317**, 2021.

RäIHä, O. A survey on search-based software design. **Computer Science Review**, v. 4, n. 4, p. 203–249, 2010.

SAAD, S. M.; BAHADORI, R.; JAFARNEJAD, H. The smart sme technology readiness assessment methodology in the context of industry 4.0. **Journal of Manufacturing Technology Management**, 2021.

SABIR, F. et al. A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems. **Software: Practice and Experience**, v. 49, n. 1, p. 3–39, 2019.

SAHIN, D. et al. Code-smell detection as a bilevel problem. **Transactions on Software Engineering and Methodology**, v. 24, n. 1, p. 1–44, 2014.

SANTOS, J. A. M. et al. A systematic review on the code smell effect. **Journal of Systems and Software**, v. 144, p. 450–477, 2018.

SARAZIN, A. et al. Toward information system architecture to support predictive maintenance approach. In: **Enterprise interoperability viii**. p. 297–306.

SCHACH, S. R. et al. Maintainability of the linux kernel. **IEE Proceedings-Software**, v. 149, n. 1, p. 18–23, 2002.

SCHMIDT, B.; WANG, L. Predictive maintenance of machine tool linear axes: a case from manufacturing industry. **Procedia manufacturing**, v. 17, p. 118–125, 2018.

SHAH, K. et al. A comparative analysis of logistic regression, random forest and knn models for the text classification. **Augmented Human Research**, v. 5, n. 1, p. 1–16, 2020.

SHARMA, T.; MISHRA, P.; TIWARI, R. Designite: a software design quality assessment tool. In: INTERNATIONAL WORKSHOP ON BRINGING ARCHITECTURAL DESIGN THINKING INTO DEVELOPERS' DAILY ACTIVITIES, 1., 2016. **Proceedings...**. p. 1–4.

SHARMA, T.; SPINELLIS, D. A survey on software smells. **Journal of Systems and Software**, v. 138, p. 158–173, 2018.

SHIPPEY, T.; BOWES, D.; HALL, T. Automatically identifying code features for software defect prediction: using ast n-grams. **Information and Software Technology**, v. 106, p. 142–160, 2019.

SJØBERG, D. I. et al. Quantifying the effect of code smells on maintenance effort. **IEEE Transactions on Software Engineering**, v. 39, n. 8, p. 1144–1156, 2012.

SJøBERG, D. I. et al. Quantifying the effect of code smells on maintenance effort. **IEEE Transactions on Software Engineering**, v. 39, n. 8, p. 1144–1156, 2013.

SOH, Z. et al. **Do code smells impact the effort of different maintenance programming activities?** 2016. 393–402 p. v. 1.

SOMMERVILLE, I. Software engineering 9th edition. **ISBN-10**, v. 137035152, p. 18, 2011.

SOUSA, B. L.; BIGONHA, M. A.; FERREIRA, K. A. A systematic literature mapping on the relationship between design patterns and bad smells. In: ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING, 33., 2018. **Proceedings...** p. 1528–1535.

SOUSA, L. et al. Identifying design problems in the source code. , 2018.

STOIAN, N.-A. **Machine learning for anomaly detection in iot networks**: malware analysis on the iot-23 data set. 2020. B.S. thesis — University of Twente, 2020.

SURYANARAYANA, G.; SAMARTHYAM, G.; SHARMA, T. **Refactoring for software design smells**: managing technical debt.

TAKAHASHI, A. et al. An extensive study on smell-aware bug localization. **Journal of Systems and Software**, v. 178, p. 110986, 2021.

TAYLOR, R. N. Software architecture and design. In: **Handbook of software engineering**. p. 93–122.

THOTA, M. K. et al. Survey on software defect prediction techniques. **International Journal of Applied Science and Engineering**, v. 17, n. 4, p. 331–344, 2020.

TOLLIN, I. et al. Change prediction through coding rules violations. In: INT. CONF. ON EVALUATION AND ASSESSMENT IN SOFTWARE ENGINEERING, 21., 2017, New York, NY, USA. **Proceedings...** p. 61–64. (EASE'17).

TSENG, M.-L. et al. Sustainable industrial and operation engineering trends and challenges toward industry 4.0: a data driven analysis. **Journal of Industrial and Production Engineering**, p. 1–18, 2021.

UCHÔA, A. et al. Predicting design impactful changes in modern code review: a large-scale empirical study. In: IEEE/ACM 18TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), 2021., 2021. **Anais...** p. 471–482.

Uchôa, A. et al. How does modern code review impact software design degradation? an in-depth empirical study. In: IEEE INT. CONF. ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), 2020., 2020. **Anais...** p. 511–522.

VIDAL, S. et al. Jspirit: a flexible tool for the analysis of code smells. In: INT. CONF. OF THE CHILEAN COMPUTER SCIENCE SOCIETY (SCCC), 2015., 2015. **Anais...** p. 1–6.

VIDAL, S. et al. **Identifying architectural problems through prioritization of code smells**. 2016. 41–50 p.

VIDAL, S. et al. On the criteria for prioritizing code anomalies to identify architectural problems. In: ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING, 31., 2016. **Proceedings. . .** p. 1812–1814.

WALTER, B.; ALKHAEIR, T. The relationship between design patterns and code smells: an exploratory study. **Information and Software Technology**, v. 74, p. 127–142, 2016.

WANG, S.; BANSAL, C.; NAGAPPAN, N. Large-scale intent analysis for identifying large-review-effort code changes. **Information and Software Technology**, v. 130, p. 106408, 2021.

WERNER, C. et al. The lack of shared understanding of non-functional requirements in continuous software engineering: accidental or essential? In: IEEE 28TH INTERNATIONAL REQUIREMENTS ENGINEERING CONFERENCE (RE), 2020., 2020. **Anais. . .** p. 90–101.

WIERINGA, R. et al. Requirements engineering paper classification and evaluation criteria: a proposal and a discussion. **Requirements engineering**, v. 11, n. 1, p. 102–107, 2006.

WŁODARSKI, L. et al. **Qualify first! a large scale modernisation report**. 2019. 569–573 p.

WOHLIN, C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: INT. CONF. ON EVALUATION AND ASSESSMENT IN SOFTWARE ENGINEERING, 18., 2014. **Anais. . .** p. 1–10.

WOHLIN, C. et al. **Experimentation in software engineering**.

WONG, W. Y.; YU, S. W.; TOO, C. W. A systematic approach to software quality assurance: the relationship of project activities within project life cycle and system development life cycle. In: CONF. ON SYSTEMS, PROCESS AND CONTROL, 2018. **Anais. . .** p. 123–128.

XU, Z. et al. Ldfr: learning deep feature representation for software defect prediction. **Journal of Systems and Software**, v. 158, p. 110402, 2019.

YAMASHITA, A.; MOONEN, L. Do code smells reflect important maintainability aspects? In: INT. CONF. ON SOFTWARE MAINTENANCE, 28., 2012. **Anais. . .** p. 306–315.

YAMASHITA, A.; MOONEN, L. Do code smells reflect important maintainability aspects? In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), 2012., 2012. **Anais. . .** p. 306–315.

ZHANG, L. et al. Odor prediction and aroma mixture design using machine learning model and molecular surface charge density profiles. **Chemical Engineering Science**, v. 245, p. 116947, 2021.

ZHOU, C.; THAM, C.-K. Graphel: a graph-based ensemble learning method for distributed diagnostics and prognostics in the industrial internet of things. In: IEEE 24TH INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED SYSTEMS (ICPADS), 2018., 2018. **Anais. . .** p. 903–909.

ZIMMERMANN, T.; PREMRAJ, R.; ZELLER, A. Predicting defects for eclipse. In: THIRD INT. WORKSHOP ON PREDICTOR MODELS IN SOFTWARE ENGINEERING (PROMISE'07: ICSE WORKSHOPS 2007), 2007. **Anais. . .** p. 9–9.

.3 Grammar *SmellDSL*

```
grammar org.smelldsl.SmellDsl with org.eclipse.xtext.common.
  Terminals

generate \textit{SmellDSL} "http://www.smelldsl.org/SmellDsl"

Domainmodel:
 (elements+=Type)*;

Type:
    SmellType | Smell | Rule;

SmellType:
    'smelltype' name=ID;


Smell:
    'smell' name=ID ('extends' superType=[SmellType])? '{'
        (smellcontent+=Feature)*
        (symptom?=Symptom)?
        (treatment?=Treatment)?
    '}';

Feature:
    'feature' name=ID (optscales?=Optscale)? 'with threshold' (
  measures+=Measure)* ;

Optscale:
     'is' (scaletype=Scaletype);

enum Scaletype:
    Nominal='Nominal' | Ordinal='Ordinal' | Interval='Interval' |
  Ratio='Ratio';


Measure: name=ID |  name=ID ',' (measure=Measure) ;

Symptom:
    'symptom' name=ID;

```

```
37 Treatment:
38     'treatment' name=ID;
39 
40 Rule:
41  'rule' name=ID 'when' (logicExpression+=LogicExpression)*  'then' (
     result+=Result)*;
42 
43 LogicExpression:
44     (expression=Expression) | (expression=Expression) (logicOperator
     =LogicOperator) (logicExpression=LogicExpression);
45 
46 enum LogicOperator:
47     AND='AND' | OR='OR';
48 
49 Expression:
50     ID('.'ID) (relationalOperator+=RelationalOperator) ID('.'ID);
51 
52 enum RelationalOperator:
53     GreaterThanOrEqualTo='>=' | LessThanOrEqualTo='<=' | NotEqualTo=
     '!=' | LessThan='<' | GreaterThan='>' | EqualTo='==';
54 
55 Result:
56  name=ID;
57 
58 }
59 
```

Listing 1: Grammar *SmellDSL*

.4 Diagram *SmellDSL*

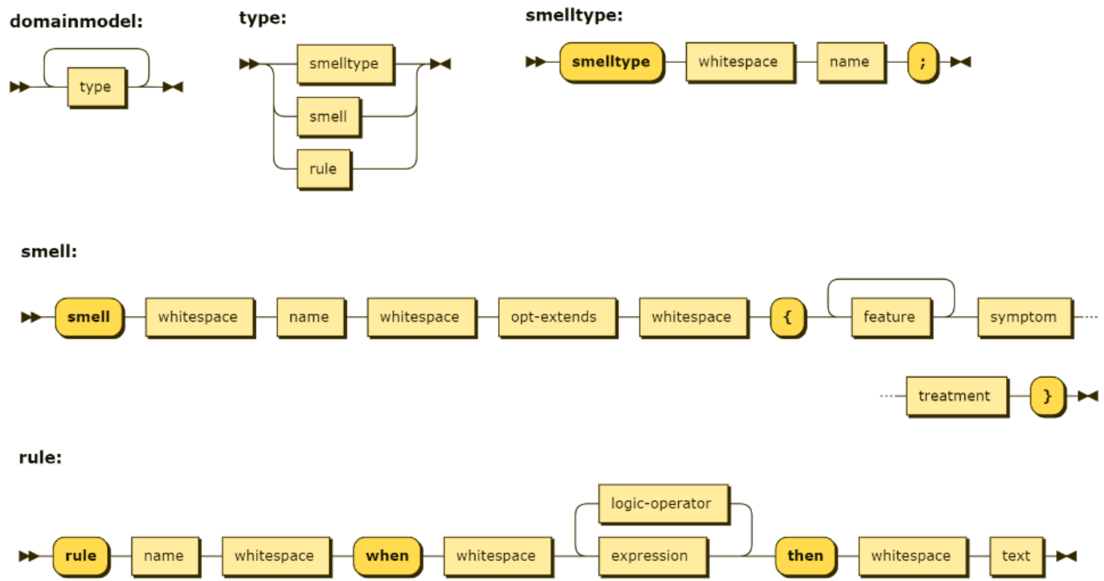

Figure 17: Diagram *SmellDSL* 01