pre-print

On the prediction of source code design problems: A systematic mapping study

R. Keemps*[1], K. Farias[1], R. Kunst[1]

[1] Applied Computing Graduate Program (PPGCA), University of Vale do Rio dos Sinos (Unisinos), Rio Grande do Sul, São Leopoldo, Brazil

Corresponding author: *robson.keemps@gmail.com

ABSTRACT

Context: 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 explored this subject in the last decade. Researchers and practitioners often need to create an overview of such studies considering the predictors of design problems, their pivotal contributions, the used prediction techniques, and research methods. *Problem:* However, the current literature lacks studies introducing a detailed mapping of published works. *Objective:* This article, therefore, aims at classifying the current literature, and pinpointing trends and challenges worth investigating in this research field. *Method:* We run a systematic mapping study following well-known guidelines. We applied a careful filtering process from a corpus of 894 candidate studies. In total, 35 primary studies were selected, analyzed, and categorized. *Results:* The main results are that a majority of the primary studies (1) explore Bloater bad smells, (2) use code complexity and size as predictors, (3) apply machine learning techniques to generate predictions, and (4) present a prediction proposal without an extensive empirical assessment. *Conclusions:* Predicting design problems is still in its infancy, showing plenty of room for future works. Finally, our findings can serve as a starting point for upcoming studies.

Keywords: Design problems, Bad smell, Systematic mapping study, Literature review, Prediction

1. Introduction

Design problems are internal structures of source code that challenge design principles or rules (Sharma et al., 2018; Oizumi et al., 2016; Suryanarayana et al., 2014). Typically, they arise when the source code of applications undergoes constant changes, for example, to accommodate new features or even evolve existing ones. These changes can lead to improper changes characterized by the scattering and tangling of software concerns, violating design principles such as the single-responsibility principle. The violation of rules and design principles gives rise to symptoms of poor design (Palomba et al., 2017) due to the presence of *code anomalies*, also popularly known as *bad smell*s (Oizumi et al., 2016).

Previous empirical studies (Oizumi et al., 2016; Palomba et al., 2018a; Palomba et al., 2017) point out that bad smells indicate design problems. Couplers and bloaters are examples of bad smells (Suryanarayana et al., 2014), which represent excessive coupling, and large proportions of source code elements, respectively. As a result, maintenance tasks become error-prone. Documenting architectural decisions through UML models (Petre, 2014; Júnior et al., 2021) could help identifying such design problems. In practice, UML models end up not being created or updated, making it difficult to identify and resolve code anomalies. As a result, developers tend to rely on tacit knowledge to prevent further violations of design principles.

In this context, predicting design problems can play an essential role, 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 on-demand. Today, the literature lacks studies that provide an overview of the state of the art on the subject of predicting

source code design problems. Having a literature mapping would help software developers to identify potential techniques as well as researchers to explore gaps in the state of the art. The absence of this mapping leads developers, for example, to adopt techniques considering the opinion of experts.

This article, therefore, proposes a systematic mapping of the literature to fill this gap. We classify the current literature and pinpoint trends and challenges worth investigating in this research field. Our mapping study follows well-known practical guidelines (Petersen et al., 2008; Petersen et al., 2015). Moreover, it is based on the authors' experience in carrying out previous mapping studies (Carbonera et al., 2020; Menzen et al., 2021; Vieira & Farias, 2020; Bischoff et al., 2021). In total, 35 primary studies were selected, analyzed, and categorized after applying a careful filtering process to a sample of 894 candidate studies to answer six research questions.

The main results show that (1) 54.29% (19/35) of the primary studies use machine learning techniques, and (2) the majority of the primary studies explored Bloaters (62.86%, 22/35), Architectural Problems (54.29%, 19/35), and Couplers (42.86%, 15/35). Our findings also indicate that predicting design problems is still in an incipient field of research, showing plenty of room for future works. Lacking techniques that can proactively anticipate the appearance of design problems, allowing for an early refactoring action to preserve the internal quality indicators of source code. Lastly, this article reports worth investigating challenges regarding the prediction of design models that can proactively influence the source code's quality. Rather than exhausting this topic, our article seeks to serve as a starting point for future investigations.

We outline the main concepts discussed throughout the article, and compare our article with similar ones, highlighting their differences and commonalities (Section 2). In addition, we present the study protocol (Section 3) and introduce the procedures adopted to filter potentially relevant studies (Section 4). After that, we present the collected results (Section 5) and outline additional discussions and future directions (Section 6). Lastly, we discuss some limitations and threats to validity (Section 7) and draw some conclusions (Section 8).

2. Background and Related Work

2.1 Design problems

Design problems are indicators of situations that improperly affect software quality attributes such as understandability, testability, extensibility, reusability, and maintainability in general. Improving maintainability is one of the cornerstones of making software evolution easier (Alkharabsheh et al., 2019). A design problem does not produce compile-time or run-time errors, but it does negatively affect the system quality attributes, such as understandability, testability, extensibility, reusability, or maintainability in general (Garcia, 2011; Yamashita et al., 2013). Design problems are indicators of refactoring opportunity (Bavota et al., 2015) or even signs that can lead to software design failure (Budgen et al., 2008).

According to Fowler and Kent (Fowler, 2018), code smells can be defined as a potential indication of problems in the source code of information systems. Thus, it is important to define new approaches to spot them whenever possible. Some systematic literature reviews (SLRs) have been conducted in the area of bad smells (Al-Shaaby et al., 2020; de Paulo et al., 2018; Fernandes et al., 2016; Lacerda et al., 2020; Rasool et al., 2015; Sabir et al., 2019; Santos et al., 2018). The code 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 (Al-Shaaby et al., 2020; Alkharabsheh et al., 2018; Di Nucci et al., 2018; Fontana et al., 2016). A machine learning classifier 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 et al., 2020).

However, the volume of research in the domain has multiplied over more than 17 years of activity in the field. This has led to the need for critical integration and evaluation of the available research in design smell detection. In our opinion, understanding this area is important because, nowadays, a

considerable number of software projects have large dimensions, so manual design smell detection is not a realistic task. Problems are latent in code; detection usually occurs very late, and then, the 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. Design smell detection is far from that reality (Alkharabsheh et al., 2019).

2.2 Prediction of design problems

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 to guess the behavior of the software in the future with respect to its coding (Kaur & Mittal, 2017; Kim et al., 2008; Palomba et al., 2018; Palomba et al., 2018b; Rani et al., 2017). Predicting the location can 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, software developers in prioritizing their work on the most severe smells (Alkharabsheh et al., 2018).

2.3 Related work

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 et al., 2019; Chen et al., 2018; Ibarra et al., 2018; Martínez-Fernández et al., 2019; Wong, 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.

Moreover, works are developed in the area of Software Analytics (Abbes et al., 2011; Barbosa et al., 2013), tooling supports, and good development practices for improved source code quality. Barbosa et al. (2020) report that software design quality degradation can be avoided, reduced, or accelerated depending on the developers' communication dynamics and the 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. Social metrics can also be indicators of design decay when analyzing the two aspects together.

It is appointed in (Uchôa et al., 2021; Uchôa et al., 2021a) that existing studies tend to analyze the degradation of the software design and consider unique events like introducing a single design problem or simply analyzing the rate of degradation. However, understanding how design degradation evolves between reviews is still challenging. The impact of modern code review on the evolution of project degradation is analyzed. Since the code review also aims to improve the quality of design and evaluations can be expected to gradually reduce over time various symptoms of software degradation. To this end, they have investigated retrospectively 14.971 code reviews of seven software systems belonging to two large open source communities. Design degradation characteristics were analyzed in revisions and within reviews. How design discussions tend to impact design degradation.

The paper (Alkharabsheh et al., 2019) presents a systematic mapping, focusing on all types of Design Smell (Bad Smell, Anti-patterns, Disharmonies, using Design Smell as a unifying term) and, on the other hand, on the detection activity and some other related activities, such as specification, correcting (refactoring), and prioritization.

Providing an overview and discussing the use of machine learning approaches in the field of bad smell 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 took machine learning approaches. These studies address four different perspectives: (i) considered code smells, (ii) configuring machine learning approaches, (iii) design of evaluation strategies, and (iv) a meta-analysis of the performance achieved by the models hitherto proposed.

The analyzes performed show that Class of God 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 large set of variables had a great 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 approaches in the field of design problems.

This paper presents a systematic mapping that differs from the previously mentioned studies by focusing, on the one hand, on all types of Code Smell and Design Smell (Bad Smell, Anti-patterns, Disharmonies, among others, using Design Smell and Code Smell as a unifying term) and, on the other hand, on the detection activity. Consequently, the main goal of our systematic mapping study is to collect and organize the knowledge on Design Smell Detection in general (approaches, tools, techniques, datasets, and quality factors) and not just tools, as is the case of some related work.

3. Planning

This section introduces an outline of the research protocol used to carry out our mapping study. Section 3.1 presents the central objective and research questions of our study. Section 3.2 defines the search strategy for selecting works already published. Section 3.3 explains how the search string was defined and the electronic databases were chosen. Section 4.4 describes the exclusion and inclusion criteria used to filter works.

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, 2007; Kitchenham et al., 2011; Petersen et al., 2008; Petersen et al., 2015). Additionally, our protocol was also inspired on our previous mapping studies (Carbonera et al., 2020; Menzen et al., 2021; Vieira & Farias, 2020; Bischoff et al., 2021)

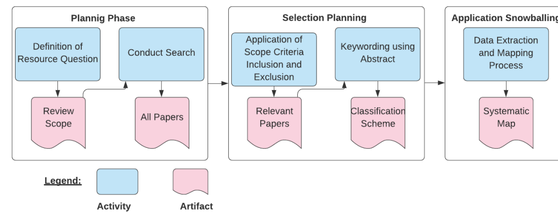

Figure 1. The systematic mapping process used in our study (adapted from (Petersen et al., 2008)).

3.1 Objective and research questions

The main objective of this mapping study is to introduce a broad and careful view of previous works considering the theme of predicting design problems. In particular, this work aims to provide an overview of the current literature by filtering (Section 4) and classify articles (Section 5), pointing out gaps, open challenges and research opportunities that are worth exploring by the scientific community (Section 6).

We formulated six research questions, described in Table 1, to address this objective. These questions seek to explore some pivotal issues, 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.

Research Question	Motivation	Variable
RQ1: What are the design problems explored by prediction techniques?	Reveal the most common explored types of design problems.	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?	Pinpoint the most used prediction techniques.	Prediction technique
RQ4: What is the main contribution of the primary studies?	Identify the explored contributions.	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

Table 1. Research questions explore in this article.

3.2 Search strategy

Based on the research questions, we determined a set of key terms to support searching for potentially relevant articles. Key terms were defined based on well-known empirical guidelines (Al-Qudah et al., 2015; Petersen et al., 2008; Wohlin, 2012). Table 2 shows the main terms and alternative terms. The selection of these terms considered their relevance to the research field, as well as strategies used and validated in previous works (Bischoff et al., 2021; Luz & Farias, 2020; Menzen et al., 2021).

Main Term	Alternative Term
Design problem	Bad smell, Code smell, code anomaly
Predict	Forecast, Foresee, Anticipate
Maintenance	Evolution, Development, Review, Refactoring

Table 2. The major terms and their alternative terms used.

3.3 Elaboration of the search string

Steps to defined the search strings. The main steps followed to define the search string were: (1) Identify candidate keywords, reading studies (e.g., (Fowler, 2018; Oizumi et al., 2016; Palomba et al., 2017; Sharma et al., 2018; Suryanarayana et al., 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 main terms using logical operators “AND”. Previous works (Petersen et al., 2015; Wohlin, 2012) 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 or design smell) 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 preparing the 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 et al., 2016; (Kitchenham, 2012; Kitchenham et al., 2011; Petersen et al., 2015; Kuutila et al., 2020), have demonstrated the effectiveness of such electronic databases used to perform literature reviews.

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

Table 3. List of the used search engine.

3.4 Exclusion and inclusion criteria

We apply a set of inclusion and exclusion criteria to the returned articles after applying the search string to the digital libraries. These criteria establish rules to make the filtering as objective and auditable as possible, avoiding biases usually found in manual tasks performed by humans. The inclusion criteria (CI) considered were:

- **CI1:** 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;
- **CI2:** Studies published between 2005 to 2020;
- **CI3:** Studies published in Portuguese or English. We agree that the most relevant research in the computing area is published in English-language vehicles, although there are exceptions; and
- **CI4:** Studies that contain key terms: Prediction, Bad Smell, Software, or Design.

The exclusion criteria (CE) considered were:

- **CE1:** The title, summary or even its content without relation to the search string;
- **CE2:** Short studies (up to 4 pages) written in another language, other than Portuguese or English;
- **CE3:** Duplicate studies;
- **CE4:** Abstract did not address any aspect of the research questions;
- **CE5:** Older versions of published studies prior to 2005;
- **CE6:** Studies that are narrowly related to Software Engineering, Software Development and/or contrary to research questions; and
- **CE7:** The full text did not address issues considering the prediction of design models;

3.5 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 et al., 2015): (1) Evaluation study: a specific problem is defined, proposing a solution and conducting 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.

4. Study Filtering

With the search string and the exclusion and inclusion criteria defined, the next step is to define the article search strategy. 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:

- **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 CE5 and CE6 were applied. For that, 38.33% (69 out of 180) of the studies were filtered.
- **Step 4: Filter by abstract.** Exclusion criteria (CE4 and CE7) 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) (Jalali & Wohlin, 2012; Wohlin, 2014). 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. The snowball method in this paper is run three times to add new selected articles.

The search was performed in the first two months of 2021. Finally, 35 studies were filtered as the most representative, hereinafter called primary studies (Table 4). The number of citations shown in Table 4 was found in Google Scholar and retrieved in January 2021.

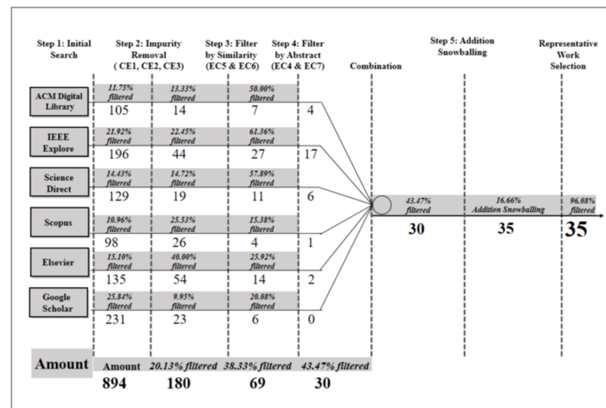

Figure 2. The filtering process.

ID	Title	Year	#Cit	#Ref
A1	Identifying Architectural Problems through Prioritization of Code Smells (Vidal et al., 2016)	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 (dos Reis et al., 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 et al., 2019)	2019	0	27
A12	Deviance from perfection is a better criterion than closeness to evil when identifying risky code (Kessentini et al., 2010)	2010	73	28
A13	Evolution of legacy system comprehensibility through automated refactoring (Griffith et al., 2011)	2011	13	30
A14	Automatically classifying source code using tree-based approaches (Petersen 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 et al., 2011)	2011	1	20
A17	Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems (Macia et al., 2012)	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 et al., 2012)	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 et al., 2019)	2019	8	81
A24	Code smell severity classification using machine learning techniques (Alkharabsheh et al., 2018)	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 et al., 2016)	2016	28	64
A28	Using (Bio)Metrics to Predict Code Quality Online (Müller et al., 2016)	2016	32	77
A29	Less is more: Minimizing code reorganization using XTREE (Krishna et al., 2017)	2017	15	68
A30	Bad-smell prediction from software design model using machine learning techniques (Maneerat et al., 2011)	2011	40	14
A31	On the criteria for prioritizing code anomalies to identify architectural problems (Vidal et al., 2016a)	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 et al., 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

Legend:
#Cit: Number of citations
#Ref: Nuber of references

Table 4. List of the selected studies.

5. Results

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

5.1 Objective and research questions

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). The classification in Table 5 is based on the work of (Mantyla et al., 2003) categorized all of Fowler's code smells except for Incomplete Library Class and Comments smells into five categories: Bloaters, Object Orientation Abusers, Change Preventers, Dispensables, Encapsulators, and Couplers. The study outlines the existence of several correlations among smells belonging to the same category. Moha & Guéhéneuc (2007) propose a taxonomy of smells and describe some correlations among design smells, such as Blob and (many) Data Class, or Blob and (Large Class and Low Cohesion). The categories of code smells we considered are based on the classification proposed in (Mäntylä et al., 2006), where the smells are classified according to some of the common concepts shared by the smells within one category.

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., 2018a; Sjøberg et al., 2012) revealed a relationship between the diffusion of bad smells and the size and complexity of the source code. Palomba et al. (2018a) point out that the smelly 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. (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 multiple code smells in classes tends to increase the change- and fault-proneness (Khomh et al., 2012; Palomba et al., 2018a), 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.

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	19/35	54.29%	[A1], [A3-7], [A9], [A11-20], [A31], [A33],
Problems			[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]

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

5.2 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 a tendency to use structural properties 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 (Moha et al., 2009; Nagappan et al., 2006; Palomba et al., 2017; Yamashita et al., 2007). 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. (2007) indicate a positive correlation between code complexity and bugs. Nagappan et al. (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 understanding 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.

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]

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

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

Table 7 introduces the collected data related to the techniques used to predict bad smell problems investigated by the selected studies. The main feature is that most primary studies used Machine Learning Techniques (54.29%, 19/35). The overall ranking accuracy of Machine Learning (ML) models is used to measure the performance of different methods and approaches. On the other hand, ML algorithms can be divided into 3 categories: supervised learning, unsupervised learning, and reinforcement learning (Chinnamgari et al., 2019; Dharmadhikari et al., 2011).

Among the selected works, we highlight the use of algorithms with a supervised learning classification and regression method. 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 essential to highlight and compare this result with the published study that notes that further studies are needed to consider the use of cluster learning, multi-classing and resource selection techniques for code smells detection (Al-Shaaby et al., 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 smell according to the classification in Table 5, 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.

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]

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

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 smells. 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.

5.4 RQ4: What is the main contribution of the primary studies?

The collected results indicate a tendency to use techniques machine learning in Table 7, used the practice of prioritizing bad smells as described in Table 5, the use of analysis 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 6 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 previous Section 5.3, in Table 8, since a classification of contribution adopted in (Petersen et al., 2008; Petersen et al., 2015), its great majority by the selected primary studies are related 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., 2015), highlight small examples are typically used to demonstrate the potential benefits and the applicability of the proposed solution.

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]

Table 8: Study classification by contributions (RQ4).

5.5 RQ5: What research methods were used?

Table 9 shows the relation between the primary studies selected and six empirical methods described in (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 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.

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]

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

5.6 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 the 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.

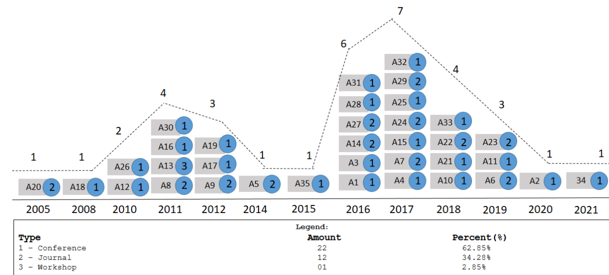

Figure 3. Distribution over the years.

6. Discussion and Future Directions

This section discusses the collected data to explore the main points of RQ5 and RQ6. In particular, we seek to reveal where the selected studies are being published over the years (Figure 3). The classification of the selected studies is based on the year of publication, publication type (workshops, conferences, newspapers, and magazines), and the number of studies published per year. Figure 4 presents the obtained data based on the 37 selected studies, showing quantitatively the results presented in RQ6.

6.1 Distribution of primary studies

Figure 4 introduces a bubble chart that organizes the 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). 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 systematic review literature to examine best practices related to source code analysis approaches, technologies or tools, and 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.

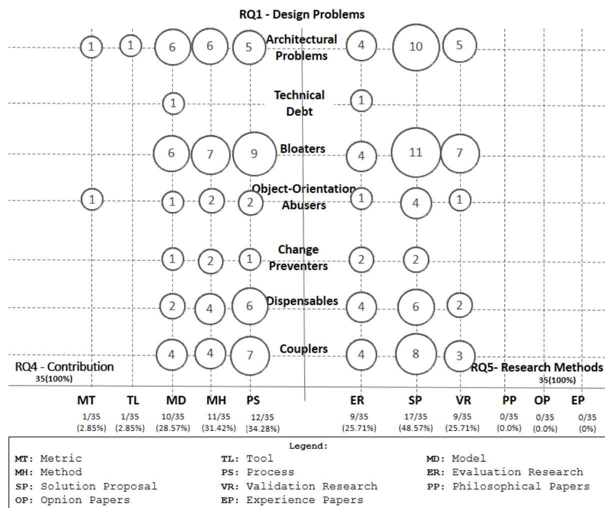

Figure 4. Bubble chart relating three variables.

6.2 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, amid a tougher economic climate, organizations turned to ML for automation and efficiency gains. This allows many of them to massively expand their operations while reallocating their human capital. 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 or techniques of ML, 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 and predict 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.

(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 data sets. The construction of a data set is challenging due to the various sources and lack of structured data. Moreover, source code only may not be sufficient to obtain good results, in artificial intelligence projects, especially Machine Learning, a large amount of data is needed, which will participate in the training of the algorithm.

So, a lot of the work on an ML project is finding the perfect data set for your needs. However, it is not always possible to find an option according to its ambition. 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 research opportunities.

7. 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 ex- traction process. In this way, studies with a broad overview were obtained.

The number of studies selected: To obtain a wide range of results and necessary data, the search 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. An interesting method to expand the possibilities of returning articles relevant to the review research topic is the snowball method. This one method consists of searching the references of articles included in the work to identify works that are of interest to the research. This method can be used, for example, at the end of the automatic search where a set of articles is already included in the review. Thus, from this set, this technique can be used to find more relevant studies.

8. Conclusions and Future Work

This article sought to grasp and classify the current literature and pinpoint trends and challenges worth investigating in the field of design problems. A systematic mapping study was designed and run based on well-established practical guidelines. In total, six research questions were formulated and answered after carefully analyzing 35 primary studies. This study fills a current gap in the literature considering the variables explored in the research questions, as well as serving as a basis for researchers and students to develop future work on the subject. Our results indicated that a majority of the primary studies explored Bloater bad smells, used code complexity and size as predictors, applied machine learning techniques to generate predictions, and presented a prediction proposal without an extensive empirical assessment.

Finally, we hope that the collected data and insights presented throughout this study can encourage researchers and practitioners to explore upcoming challenges regarding the prediction of design problems. Moreover, this work can be seen as the first step for a more robust agenda on how to advance the current literature on the prediction of design problems.

Acknowledgements

This work was partially supported by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) under Grant 314248/2021-8.

References

Abbes, M., Khomh, F., Gueheneuc, Y. G., & Antoniol, G. (2011, March). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In 15th European conference on software maintenance and reengineering (pp. 181-190). IEEE.

Al-Qudah, S., Meridji, K., & Al-Sarayreh, K. T. (2015, November). A comprehensive survey of software development cost estimation studies. In International conference on intelligent information processing, security and advanced communication (pp. 1-5).

Al-Shaaby, A., Aljamaan, H., & Alshayeb, M. (2020). Bad smell detection using machine learning techniques: a systematic literature review. Arabian Journal for Science and Engineering, 45(4), 2341-2369.

Alenezi, M., Akour, M., Hussien, A., & Al-Saad, M. Z. (2016, December). Test suite effectiveness: an indicator for open source software quality. In 2016 2nd International

Conference on Open Source Software Computing (OSSCOM) (pp. 1-5). IEEE.

Alkharabsheh, K., Crespo, Y., Manso, E., & Taboada, J. A. (2019). Software design smell detection: a systematic mapping study. Software Quality Journal, 27(3), 1069-1148.

Alkharabsheh, K., Taboada, J. A., Crespo, Y., & Alzu'bi, T. (2018, July). Improving design smell detection for adoption in industry. In 2018 8th International Conference on Computer Science and Information Technology (CSIT) (pp. 213-218). IEEE.

Azeem, M. I., Palomba, F., Shi, L., & Wang, Q. (2019). Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. Information and Software Technology, 108, 115-138.

Barbosa, C., Uchôa, A., Coutinho, D., Falcão, F., Brito, H., Amaral, G., ... & Sousa, L. (2020, October). Revealing the social aspects of design decay: A retrospective study of pull requests. In Proceedings of the 34th Brazilian Symposium on Software Engineering (pp. 364-373).

Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., & Palomba, F. (2015). An experimental investigation on the innate relationship between quality and refactoring. Journal of Systems and Software, 107, 1-14.

Boehm, B., Rosenberg, D., & Siegel, N. (2019, July). Critical quality factors for rapid, scalable, agile development. In 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C) (pp. 514-515). IEEE.

Boussaa, M., Kessentini, W., Kessentini, M., Bechikh, S., & Ben Chikha, S. (2013, August). Competitive coevolutionary code-smells detection. In International Symposium on Search Based Software Engineering (pp. 50-65). Springer, Berlin, Heidelberg.

Bischoff, V., Farias, K., Menzen, J. P., & Pessin, G. (2021). Technological support for detection and prediction of plant diseases: A systematic mapping study. Computers and Electronics in Agriculture, 181, 105922.

Budgen, D., Bailey, J., Turner, M., Kitchenham, B., Brereton, P., & Charters, S. (2008, June). Lessons from a cross-domain investigation of empirical practices. In 12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12 (pp. 1-12).

Carbonera, C. E., Farias, K., & Bischoff, V. (2020). Software development effort estimation: A systematic mapping study. IET Software, 14(4), 328-344.

Chen, C., Lin, S., Shoga, M., Wang, Q., & Boehm, B. (2018, July). How do defects hurt qualities? an empirical study on characterizing a software maintainability ontology in open source software. In 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS) (pp. 226-237). IEEE.

Chinnamgari, S. K. (2019). R Machine Learning Projects: Implement supervised, unsupervised, and reinforcement learning techniques using R 3.5. Packt Publishing Ltd.

Das, A. K., Yadav, S., & Dhal, S. (2019, October). Detecting code smells using deep learning. In TENCON 2019-2019 IEEE Region 10 Conference (TENCON) (pp. 2081-2086). IEEE.

de Paulo Sobrinho, E. V., De Lucia, A., & de Almeida Maia, M. (2018). A systematic literature review on bad smells–5 w's: which, when, what, who, where. IEEE Transactions on Software Engineering, 47(1), 17-66.

Devi, D. G., & Punithavalli, M. (2011, April). A hierarchical method for detecting codeclone. In 2011 3rd International Conference on Electronics Computer Technology (Vol. 1, pp. 126-128). IEEE.

Dharmadhikari, S. C., Ingle, M., & Kulkarni, P. (2011). Empirical studies on machine learning based text classification algorithms. Advanced Computing, 2(6), 161.

Di Nucci, D., Palomba, F., Tamburri, D. A., Serebrenik, A., & De Lucia, A. (2018, March). Detecting code smells using machine learning techniques: are we there yet?. In 2018 ieee 25th international conference on software analysis, evolution and reengineering (saner) (pp. 612-621). IEEE.

dos Reis, J. P., e Abreu, F. B., & Carneiro, G. D. F. (2017, June). Code smells detection 2.0: Crowdsmelling and visualization. In 2017 12th Iberian Conference on Information Systems and Technologies (CISTI) (pp. 1-4). IEEE.

El Koutbi, S., Idri, A., & Abran, A. (2016, August). Systematic mapping study of dealing with error in software development effort estimation. In 2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (pp. 140-147). IEEE.

Erturk, E., & Sezer, E. A. (2016). Iterative software fault prediction with a hybrid approach. Applied Soft Computing, 49, 1020-1033.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., & Figueiredo, E. (2016, June). A review-based comparative study of bad smell detection tools. In Proceedings of the

20th International Conference on Evaluation and Assessment in Software Engineering (pp. 1-12).

Fontana, F. A., & Zanoni, M. (2017). Code smell severity classification using machine learning techniques. Knowledge-Based Systems, 128, 43-58.

Arcelli Fontana, F., Mäntylä, M. V., Zanoni, M., & Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. Empirical Software Engineering, 21(3), 1143-1191.

Fowler, M. (2018). Refactoring: improving the design of existing code. Addison-Wesley Professional.

Garcia, J. P. (2011). Refactoring planning for design smell correction in Object-oriented software.

Griffith, I., Wahl, S., & Izurieta, C. (2011, November). Evolution of legacy system comprehensibility through automated refactoring. In Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering (pp. 35-42).

Hadj-Kacem, M., & Bouassida, N. (2018, March). A Hybrid Approach To Detect Code Smells using Deep Learning. In ENASE (pp. 137-146).

Ibarra, S., & Muñoz, M. (2018, October). Support tool for software quality assurance in software development. In 2018 7th International Conference On Software Process Improvement (CIMPS) (pp. 13-19). IEEE.

Irwanto, D. (2010, December). Visual Indicator Component Software to Show Component Design Quality and Characteristic. In 2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies (pp. 50-54). IEEE.

Jalali, S., & Wohlin, C. (2012, September). Systematic literature studies: database searches vs. backward snowballing. In Proceedings of the 2012 ACM-IEEE international symposium on empirical software engineering and measurement (pp. 29-38). IEEE.

Júnior, E., Farias, K., & Silva, B. (2021, September). A Survey on the Use of UML in the Brazilian Industry. In Brazilian Symposium on Software Engineering (pp. 275-284).

Kaur, P., & Mittal, P. (2017). Impact of Clones Refactoring on External Quality Attributes of Open Source Softwares. International Journal of Advanced Research in Computer Science, 8(5).

Keele, S. (2007). Guidelines for performing systematic literature reviews in software engineering (Vol. 5). Technical report, Ver. 2.3 EBSE Technical Report. EBSE.

Kessentini, M., & Ouni, A. (2017, May). Detecting android smells using multi-objective genetic programming. In 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft) (pp. 122-132). IEEE.

Kessentini, M., Vaucher, S., & Sahraoui, H. (2010, September). Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In Proceedings of the IEEE/ACM international conference on Automated software engineering (pp. 113-122).

Khomh, F., Penta, M. D., Guéhéneuc, Y. G., & Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change-and fault-proneness. Empirical Software Engineering, 17(3), 243-275.

Khomh, F., Vaucher, S., Guéhéneuc, Y. G., & Sahraoui, H. (2011). BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. Journal of Systems and Software, 84(4), 559-572.

Kim, S., Whitehead, E. J., & Zhang, Y. (2008). Classifying software changes: Clean or buggy?. IEEE Transactions on Software Engineering, 34(2), 181-196.

Kitchenham, B. A. (2012, September). Systematic review in software engineering: where we are and where we should be going. In Proceedings of the 2nd international workshop on Evidential assessment of software technologies (pp. 1-2).

Kitchenham, B. A., Budgen, D., & Brereton, O. P. (2011). Using mapping studies as the basis for further research–a participant-observer case study. Information and Software Technology, 53(6), 638-651.

Kreimer, J. (2005). Adaptive detection of design flaws. Electronic Notes in Theoretical Computer Science, 141(4), 117-136.

Krishna, R., Menzies, T., & Layman, L. (2017). Less is more: Minimizing code reorganization using XTREE. Information and Software Technology, 88, 53-66.

Kuutila, M., Mäntylä, M., Farooq, U., & Claes, M. (2020). Time pressure in software engineering: A systematic review. Information and Software Technology, 121, 106257.

Lacerda, G., Petrillo, F., Pimenta, M., & Guéhéneuc, Y. G. (2020). Code smells and refactoring: A tertiary systematic review of challenges and observations. Journal of Systems and Software, 167, 110610.

Lenarduzzi, V., Lomio, F., Huttunen, H., & Taibi, D. (2020, February). Are sonarqube rules inducing bugs?. In 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 501-511). IEEE.

Li, J., He, P., Zhu, J., & Lyu, M. R. (2017, July). Software defect prediction via convolutional neural network. In 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS) (pp. 318-328). IEEE.

Luz, M. A. D., & Farias, K. (2020, November). The Use of Blockchain in Financial Area: A Systematic Mapping Study. In XVI Brazilian Symposium on Information Systems (pp. 1-8).

Liu, H., Ma, Z., Shao, W., & Niu, Z. (2011). Schedule of bad smell detection and resolution: A new way to save effort. IEEE transactions on Software Engineering, 38(1), 220-235.

Macia, I., Garcia, J., Popescu, D., Garcia, A., Medvidovic, N., & von Staa, A. (2012, March). Are automatically-detected code anomalies relevant to architectural modularity? An exploratory analysis of evolving systems. In Proceedings of the 11th annual international conference on Aspect-oriented Software Development (pp. 167-178).

Maneerat, N., & Muenchaisri, P. (2011, May). Bad-smell prediction from software design model using machine learning techniques. In 2011 Eighth international joint conference on computer science and software engineering (JCSSE) (pp. 331-336). IEEE.

Mantyla, M., Vanhanen, J., & Lassenius, C. (2003, September). A taxonomy and an initial empirical study of bad smells in code. In International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings. (pp. 381-384). IEEE.

Mäntylä, M. V., & Lassenius, C. (2006). Subjective evaluation of software evolvability using code smells: An empirical study. Empirical Software Engineering, 11(3), 395-431.

Martínez-Fernández, S., Vollmer, A. M., Jedlitschka, A., Franch, X., López, L., Ram, P., ... & Partanen, J. (2019). Continuously assessing and improving software quality with software analytics tools: a case study. IEEE access, 7, 68219-68239.

Menzen, J. P., Farias, K., & Bischoff, V. (2021). Using biometric data in software engineering: a systematic mapping study. Behaviour & Information Technology, 40(9), 880-902.

Moha, N., & Guéhéneuc, Y. G. (2007, November). Decor: a tool for the detection of design defects. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (pp. 527-528).

Moha, N., Guéhéneuc, Y. G., Duchien, L., & Le Meur, A. F. (2009). Decor: A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering, 36(1), 20-36.

Müller, S. C., & Fritz, T. (2016, May). Using (bio) metrics to predict code quality online. In 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE) (pp. 452-463). IEEE.

Mumtaz, H., Alshayeb, M., Mahmood, S., & Niazi, M. (2018). An empirical study to improve software security through the application of code refactoring. Information and Software Technology, 96, 112-125.

Nagappan, N., Ball, T., & Zeller, A. (2006, May). Mining metrics to predict component failures. In Proceedings of the 28th international conference on Software engineering (pp. 452-461).

Oizumi, W., Garcia, A., Sousa, L. D. S., Cafeo, B., & Zhao, Y. (2016, May). Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE) (pp. 440-451). IEEE.

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., & De Lucia, A. (2018). A large-scale empirical study on the lifecycle of code smell co-occurrences. Information and Software Technology, 99, 1-10.

Palomba, F., Bavota, G., Penta, M. D., Fasano, F., Oliveto, R., & Lucia, A. D. (2018). On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empirical Software Engineering, 23(3), 1188-1221.

Palomba, F., Zaidman, A., & De Lucia, A. (2018, September). Automatic test smell detection using information retrieval techniques. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 311-322). IEEE.

Palomba, F., Zanoni, M., Fontana, F. A., De Lucia, A., & Oliveto, R. (2017). Toward a smell-aware bug prediction model. IEEE Transactions on Software Engineering, 45(2), 194-218.

Petersen, K., Feldt, R., Mujtaba, S., & Mattsson, M. (2008, June). Systematic mapping studies in software engineering. In 12th International Conference on

Evaluation and Assessment in Software Engineering (EASE) 12 (pp. 1-10).

Petersen, K., Vakkalanka, S., & Kuzniarz, L. (2015). Guidelines for conducting systematic mapping studies in software engineering: An update. Information and software technology, 64, 1-18.

Petre, M. (2014). “No shit” or “Oh, shit!”: responses to observations on the use of UML in professional practice. Software & Systems Modeling, 13(4), 1225-1235.

Phan, A. V., Chau, P. N., Le Nguyen, M., & Bui, L. T. (2018). Automatically classifying source code using tree-based approaches. Data & Knowledge Engineering, 114, 12-25.

Plosch, R., Gruber, H., Hentschel, A., Pomberger, G., & Schiffer, S. (2008, October). On the relation between external software quality and static code analysis. In 2008 32nd Annual IEEE Software Engineering Workshop (pp. 169-174). IEEE.

Prähofer, H., Angerer, F., Ramler, R., & Grillenberger, F. (2016). Static code analysis of IEC 61131-3 programs: Comprehensive tool support and experiences from large-scale industrial application. IEEE Transactions on Industrial Informatics, 13(1), 37-47.

Rani, A., & Chhabra, J. K. (2017, April). Evolution of code smells over multiple versions of softwares: An empirical investigation. In 2017 2nd International Conference for Convergence in Technology (I2CT) (pp. 1093-1098). IEEE.

Rasool, G., & Arshad, Z. (2015). A review of code smell mining techniques. Journal of Software: Evolution and Process, 27(11), 867-895.

Sabir, F., Palma, F., Rasool, G., Guéhéneuc, Y. G., & Moha, N. (2019). A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems. Software: Practice and Experience, 49(1), 3-39.

Sahin, D., Kessentini, M., Bechikh, S., & Deb, K. (2014). Code-smell detection as a bilevel problem. ACM Transactions on Software Engineering and Methodology (TOSEM), 24(1), 1-44.

Santos, J. A. M., Rocha-Junior, J. B., Prates, L. C. L., do Nascimento, R. S., Freitas, M. F., & de Mendonça, M. G. (2018). A systematic review on the code smell effect. Journal of Systems and Software, 144, 450-477.

Sharma, T., & Spinellis, D. (2018). A survey on software smells. Journal of Systems and Software, 138, 158-173.

Shippey, T., Bowes, D., & Hall, T. (2019). Automatically identifying code features for software defect prediction: Using AST N-grams. Information and Software Technology, 106, 142-160.

Sjøberg, D. I., Yamashita, A., Anda, B. C., Mockus, A., & Dybå, T. (2012). Quantifying the effect of code smells on maintenance effort. IEEE Transactions on Software Engineering, 39(8), 1144-1156.

Soh, Z., Yamashita, A., Khomh, F., & Guéhéneuc, Y. G. (2016, March). Do code smells impact the effort of different maintenance programming activities?. In 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER) (Vol. 1, pp. 393-402). IEEE.

Suryanarayana, G., Samarthyam, G., & Sharma, T. (2014). Refactoring for software design smells: managing technical debt. Morgan Kaufmann.

Tollin, I., Fontana, F. A., Zanoni, M., & Roveda, R. (2017, June). Change prediction through coding rules violations. In Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering (pp. 61-64).

Uchôa, A., Barbosa, C., Coutinho, D., Oizumi, W., Assunçao, W. K., Vergilio, S. R., ... & Garcia, A. (2021, May). Predicting design impactful changes in modern code review: A large-scale empirical study. In 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) (pp. 471-482). IEEE.

Uchôa, A., Barbosa, C., Oizumi, W., Blenílio, P., Lima, R., Garcia, A., & Bezerra, C. (2020, September). How does modern code review impact software design degradation? an in-depth empirical study. In 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 511-522). IEEE.

Vidal, S., Guimaraes, E., Oizumi, W., Garcia, A., Pace, A. D., & Marcos, C. (2016, September). Identifying architectural problems through prioritization of code smells. In 2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS) (pp. 41-50). IEEE.

Vidal, S., Guimaraes, E., Oizumi, W., Garcia, A., Pace, A. D., & Marcos, C. (2016, April). On the criteria for prioritizing code anomalies to identify architectural problems. In Proceedings of the 31st Annual ACM Symposium on Applied Computing (pp. 1812-1814).

Vidal, S., Vazquez, H., Diaz-Pace, J. A., Marcos, C., Garcia, A., & Oizumi, W. (2015, November). JSpIRIT: a flexible tool for the analysis of code smells. In 2015 34th

International Conference of the Chilean Computer Science Society (SCCC) (pp. 1-6). IEEE.

Vieira, R. D., & Farias, K. (2020, November). Usage of psychophysiological data as an improvement in the context of software engineering: A systematic mapping study. In XVI Brazilian Symposium on Information Systems (pp. 1-8).

Wieringa, R., Maiden, N., Mead, N., & Rolland, C. (2006). Requirements engineering paper classification and evaluation criteria: a proposal and a discussion. Requirements engineering, 11(1), 102-107.

Wohlin, C. (2014, May). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In Proceedings of the 18th international conference on evaluation and assessment in software engineering (pp. 1-10).

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). Experimentation in software engineering. Springer Science & Business Media.

Wong, W. Y., Yu, S. W., & Too, C. W. (2018, December). A systematic approach to software quality assurance: the relationship of project activities within project life cycle and system development life cycle. In 2018 IEEE Conference on Systems, Process and Control (ICSPC) (pp. 123-128). IEEE.

Xu, Z., Li, S., Xu, J., Liu, J., Luo, X., Zhang, Y., ... & Tang, Y. (2019). LDFR: Learning deep feature representation for software defect prediction. Journal of Systems and Software, 158, 110402.

Yamashita, A., & Counsell, S. (2013). Code smells as system-level indicators of maintainability: An empirical study. Journal of Systems and Software, 86(10), 2639-2653.

Yamashita, A., & Moonen, L. (2012, September). Do code smells reflect important maintainability aspects?. In 2012 28th IEEE international conference on software maintenance (ICSM) (pp. 306-315). IEEE.

Zimmermann, T., Premraj, R., & Zeller, A. (2007, May). Predicting defects for eclipse. In Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007) (pp. 9-9). IEEE.