A Survey on Software Smells

Tushar Sharma and Diomidis Spinellis

Department of Management Science and Technology
Athens University of Economics and Business
{*tushar,dds*}*@aueb.gr*

Abstract

Context: Smells in software systems impair software quality and make them hard to maintain and evolve. The software engineering community has explored various dimensions concerning smells and produced extensive research related to smells. The plethora of information poses challenges to the community to comprehend the state-of-the-art tools and techniques.
Objective: We aim to present the current knowledge related to software smells and identify challenges as well as opportunities in the current practices.
Method: We explore the definitions of smells, their causes as well as effects, and their detection mechanisms presented in the current literature. We studied 445 primary studies in detail, synthesized the information, and documented our observations.
Results: The study reveals five possible defining characteristics of smells — *indicator, poor solution, violates best-practices, impacts quality,* and *recurrence*. We curate ten common factors that cause smells to occur including *lack of skill or awareness* and *priority to features over quality*. We classify existing smell detection methods into five groups — *metrics, rules/heuristics, history, machine learning,* and *optimization-based detection*. Challenges in the smells detection include the *tools' proneness to false-positives* and *poor coverage of smells detectable by existing tools*.

Keywords: Code smells, Software smells, Antipatterns, Software quality, Maintainability, Smell detection tools, Technical debt

1. Introduction

Kent Beck coined the term "code smell" in the context of identifying quality issues in code that can be refactored to improve the maintainability of a software [38]. He emphasized that the presence of excessive number of smells in a software system makes the software hard to maintain and evolve.

Since then, the smell metaphor has been extended to various related subdomains of software systems including testing [26], database [52], and configuration [109]. Further, since the inception of the metaphor, the software engineering community has explored various associated dimensions that include proposing a catalog of smells, detecting smells using a variety of techniques, exploring the relationships among smells, and identifying the causes and effects of smells.

The large number of available resources poses a challenge, equally to both researchers and practitioners, to comprehend the status quo of tools, methods, and techniques concerning software smells. Analyzing and synthesizing available information could not only help the software engineering community understand the existing knowledge, but also reveal the challenges that exist in the present set of methods and opportunities to address them.

There have been a few attempts to understand current practices and provide an overview of the existing knowledge about software smells. Singh et al. [116] present a systematic literature review on code smells and refactoring in object-oriented software systems by studying 238 primary studies. The survey focuses on the smell detection methods and tools as well as the techniques and tools used to refactor them. The authors divide smell detection methods based on the degree of automation employed in smell detection methods.

Similarly, Zhang et al. [140] review studies from year 2000 to 2009 and draw a few observations about current research on smells. They reveal a large gap in existing smells' literature — current studies have chosen a small number of smells for their study and some of the smells (such as message chains) are poorly explored by the community. Further, the study emphasizes that the impact of code smells is not well understood.

Various tools have been proposed to detect smells. Fernandes et al. [32] provides a comprehensive study containing a comparison of 84 smell detection tools. Similarly, Rasool et al. [103] also review existing code smell detection tools and reveal the challenges associated with them. A few studies [4, 78] provide an extensive coverage to refactoring techniques available to refactor

the smells.

In this study, we explore the resources related to smells extensively that were published between the years 1999 – 2016 and present the current knowledge in a synthesized and consolidated form. Additionally, our goal is to identify challenges in the present knowledge and find opportunities to overcome them.

1.1. Contributions of this study

This survey makes the following contributions to the field.

- The study provides a holistic status quo of various dimensions associated with software smells. These dimensions include definition, classification, types, detection methods, as well as causes and impacts of smells.
- It presents the state-of-the-art in the current research, reveal deficiencies in present tools and techniques, and identifies research opportunities to advance the domain.

The rest of the paper is organized as follows: we define the methodology followed in the study in Section 2. We discuss the results in Section 3 and we present our conclusions in Section 4.

2. Methodology

In this section, we first present the objectives of this study and derived research questions. We illustrate the search protocol that we used to identify relevant studies. The search protocol includes not only the steps to collect the initial set of studies, but also inclusion and exclusion criteria that we apply on the initial set of studies to obtain a filtered set of primary studies.

2.1. Research objectives and questions

The goal of this study is to provide a consolidated yet extensive overview of software smells covering their definition, types, causes, detection methods, and impact on various aspects of software development. In this study, we address the following research questions:

RQ1 *What is the definition of a software smell?*
We aim to understand how the term "smell" is defined by various researchers. We infer basic defining characteristics and types of smells.

RQ2 *How do smells get introduced in software systems?*
We explore the reasons that cause smells in software systems.

RQ3 *How do smells affect the software development processes, artifacts, or people?*
We present the impact of smells on software systems. Specifically, we study impacts of smells on processes, artifacts, and people.

RQ4 *How do smells get detected?*
We discuss the techniques employed by researchers to identify smells.

RQ5 *What are the open research questions?*
We present the perceived deficiencies and the open research questions with respect to smells, their detection, and their interpretations.

2.2. Literature search protocol

The literature search protocol aims to identify primary studies which form the basis of the survey. Our search protocol has three phases:

1. We identify a list of relevant conferences and journals and manually search their proceedings.
2. We search seven well-known digital libraries.
3. We perform filtering and consolidation of the studies identified in the previous phases and prepare a single library of relevant studies.

2.2.1. Literature search – Phase 1

We identify a comprehensive list of conferences and journals based on papers published in these venues related to smells. We manually search the proceedings of the selected venues between the year 1999 and 2016. The start year has been selected as 1999 since the smell metaphor was introduced in 1999. During the manual search, the following set of terms were searched in the title of studies: *smell, antipattern, quality, maintainability, maintenance, and metric*. All the studies containing at least one of the search terms in their title were selected and recorded. Table 1 presents the selected conferences and journals along with their corresponding number of studies selected in Phase 1.

The domain of refactoring is closely related to that of software smells. Given the vast knowledge present in the field of refactoring, it requires a separate study specifically for software refactoring. Therefore, we consider work concerning refactoring outside the scope of this study.

Table 1: Studies selected in the Phase 1

Venue	Type	#Studies
Automated Software Engineering	Conference	24
Empirical Software Engineering	Journal	61
Empirical Software Engineering and Measurement	Conference	68
European Conference on Object-Oriented Programming	Conference	2
Foundations of Software Engineering	Conference	19
IEEE Software	Journal	78
International Conference of Software Maintenance and Evolution	Conference	220
International Conference on Program Comprehension	Conference	38
International Conference on Software Engineering	Conference	85
Journal of Systems and Software	Journal	146
Mining Software Repositories	Conference	28
Software Analysis, Evolution, and Reengineering / European Conference on Software Maintenance and Reengineering	Conference	135
Source Code Analysis and Manipulation	Conference	22
Systems, Programming, Languages and Applications: Software for Humanity	Conference	8
Transactions on Software Engineering	Journal	83
Transactions on Software Engineering and Methodology	Journal	11
Total selected studies in Phase 1		**1028**

2.2.2. Literature search – Phase 2

In the second phase, we carried out search on seven well-known digital libraries. The terms used for the search are: *software smell, antipattern, software quality, maintainability, maintenance,* and *software metric*. We appended the term "software" to the search terms in order to obtain more relevant results. Additionally, we apply filters such as "computer science" and "software engineering" wherever it was possible and relevant to refine the search results. Table 2 shows the searched digital libraries and corresponding number of selected studies.

Table 2: Studies selected in the Phase 2

Digital Library	Number of studies
Google Scholar	196
SpringerLink	44
ACM Digital Library	108
ScienceDirect	40
Scopus	150
IEEE Xplore	151
Web of Science	58
Total selected studies in Phase 2	**747**

2.2.3. Literature search – Phase 3

In the third phase, we defined inclusion and exclusion criteria to filter out irrelevant studies and to prepare a consolidated library. The inclusion/exclusion criteria are listed below.

Inclusion criteria

- Studies that discuss smells in software development, present a catalog of one of the different types of software smells (such as code smells, test smells, and configuration smells), produce factors that cause smells, or explore their impact on any facet of software development (for instance, artifacts, people, or process).
- Studies introducing smell detection mechanisms or providing a comparison using any suitable technique.
- Resources revealing the deficiencies in the present set of methods, tools, and practices.

Exclusion criteria

- Studies focusing on external (in-use) software quality or not directly related with software smells.
- Studies that propose the refactoring of smells, or identifies refactoring opportunities.
- Articles containing keynote, extended abstract, editorial, tutorial, poster, or panel discussion (due to insufficient details and small size).

- Studies whose full text is not available.

Each selected article from phase 1 or 2 went through a manual inspection of title, keywords, and abstract. The inspection applied the inclusion and exclusion criteria leading to inclusion or exclusion of the articles. We obtained 445 articles after completing the inspection and removing the duplicates. These articles are the primary studies that we studied in detail. We took notes while studying the selected articles. We then mark all the relevant articles for each research question and included them in the corresponding discussion.

We did not limit ourselves to only to the primary studies. We included secondary sources of information and articles as and when we spotted them while studying primary studies. Therefore, although our primary studies belong to the period 1999 – 2016, due to the inclusion of the secondary studies, we refer studies in this paper that were published before or after the selected period. An interested reader may find the list of all the selected papers in each phase online [112].

After we completed the detailed study, we categorized the resources based on the dimensions of smells they belong to. Figure 1 provides an overview of the studied dimensions of software smells; a number in brackets shows the number of associated references. A document containing extended version of the figure has been made available online [108]; the document additionally shows all the references in the figure.

3. Results and Discussion

In this section, we present our synthesized observations corresponding to each research question addressed in this study.

3.1. RQ1: What is the definition of a software smell?

We break down the question into the following sub-questions where each sub-question deals with precisely one specific aspect of software smells' definition.

RQ1.1 *What are the defining characteristics of a software smell?*

RQ1.2 *What are the types of smells?*

RQ1.3 *How are the smells classified?*

RQ1.4 *Are smells and antipatterns considered synonyms?*

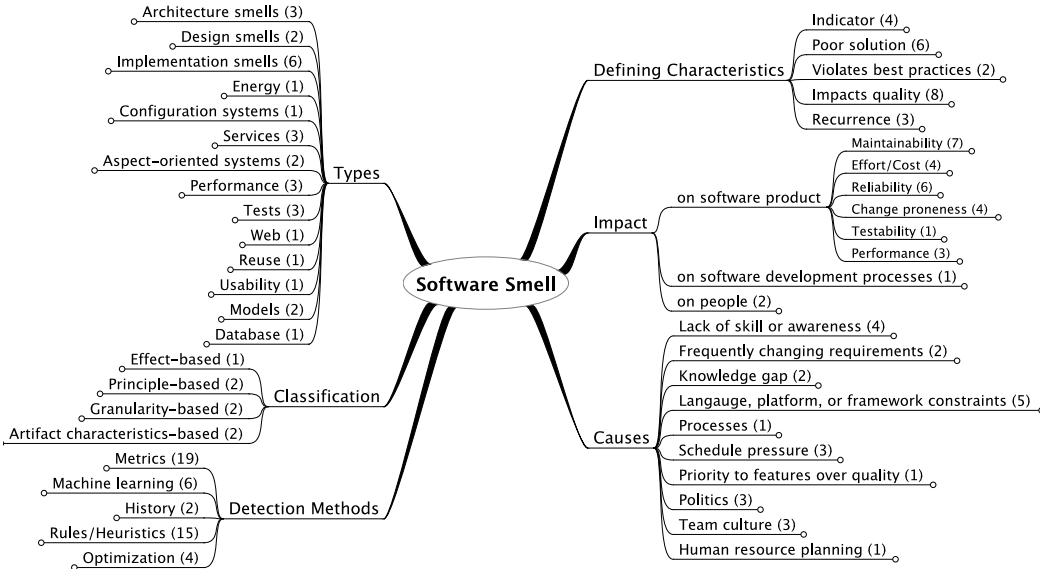

Figure 1: Overview of the study; a number in brackets shows the number of associated references

3.1.1. RQ1.1: What are the defining characteristics of a software smell?

Kent Beck coined the term "code smell" [38] and defined it informally as *"certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring"*. Later, various researchers gave diverse definitions of software smells. A complete list of definitions of smells provided by various authors can be found online [111]. Based on these, we synthesize the following five possible defining characteristics of a software smell.

- **Indicator:** Authors define smells as an indicator to or a symptom of a deeper design problem [79, 136, 123, 24].
- **Poor solution:** The literature describes smells as a suboptimal or poor solution [58, 54, 37, 9, 129, 20].
- **Violates best practices:** According to authors such as Suryanarayana et al. [123] and Sharma et al. [109], smells violate recommended best practices of the domain.
- **Impacts quality:** Smells make it difficult for a software system to evolve and maintain [136, 58]. It is commonly agreed that smells impact the quality of the system [51, 80, 9, 42, 109, 123].

- **Recurrence:** Many authors define smells as recurring problems [70, 101, 58].

3.1.2. RQ1.2: What are the types of smells?

Authors have explored smells in different domains and in different focus areas. Within software system domain, authors have focused on specific aspects such as configuration systems, tests, and models. These explorations have resulted in various smell catalogs. Table 3 presents a summary of catalogs and corresponding references.

Table 3: Types of smells

Focus	**References**
Implementation	[38], [9], [16], [29] [1], [46]
Design	[123] [15]
Architecture	[42], [16] [62]
Tests	[45], [49] [26]
Performance	[118], [114], [135]
Configuration systems	[109]
Database	[52]
Aspect-oriented systems	[6], [14]
Energy	[131]
Models	[28], [25]
Services	[92], [61], [94]
Usability	[5]
Reuse	[67]
Web	[86]

We have compiled an extensive catalog belonging to each focus area. Here, considering the space constraints, we provide a brief catalog of code smells in Table 4. We have selected the smells to include in this table based on the popularity of the smells *i.e.,* based on the number of times the smell has been studied in the literature. The comprehensive and evolving taxonomy of software smells can be accessed online.[1]

[1] `http://www.tusharma.in/smells`

Table 4: Common code smells

Code Smell / References	**Description**
God class [104]	The god class smell occurs when a huge class which is surrounded by many data classes acts as a controller (*i.e.,* takes most of the decisions and monopolises the functionality offered by the software). The class defines many data members and methods and exhibits low cohesion. **Related smells:** Insufficient modularization [123], Blob [16], Brain class [133].
Feature envy[38]	This smell occurs when a method seems more interested in a class other than the one it actually is in.
Shotgun surgery [38]	This smell characterizes the situation when one kind of change leads to a lot of changes to multiple different classes. When the changes are all over the place, they are hard to find, and it's easy to miss an important change.
Data class [38]	This smell occurs when a class contains only fields and possibly getters/setters without any behavior (methods). **Related smells:** Broken modularization [123].
Long method[38]	This smell occurs when a method is too long to understand. **Related smells:** God method [104], Brain method [133].
Functional decomposition[16]	This smell occurs when the experienced developers coming from procedural languages background write highly procedural and non-object-oriented code in an object-oriented language.
Refused bequest [38]	This smell occurs when a subclass rejects some of the methods or properties offered by its superclass. **Related smells:** Rebellious hierarchy [123]
Spaghetti code [16]	This smell refers to an unmaintainable, incomprehensible code without any structure. The smell does not exploit and prevents the use of object-orientation mechanisms and concepts.

Table 4: Common code smells

Code Smell / References	**Description**
Divergent change [38]	Divergent change occurs when one class is commonly changed in different ways for different reasons. **Related smells:** Multifaceted abstraction [123].
Long parameter list[38]	This smell occurs when a method accepts a long list of parameters. Such lists are hard to understand and difficult to use.
Duplicate code [38]	This smell occurs when same code structure is duplicated to multiple places within a software system. **Related smells:** Duplicate abstraction [123], Unfactored hierarchy [123], Cut and paste programming [16].
Cyclically-dependent modularization [123]	This smell arises when two or more abstractions depend on each other directly or indirectly. **Related smells:** Dependency cycles [76]
Deficient encapsulation [123]	This smell occurs when the declared accessibility of one or more members of an abstraction is more permissive than actually required. **Related smells:** Excessive global variables [30].
Lava flow [16]	This smell is characterized by a piece of code that nobody remembers the purpose and usage, and is largely not utilized. **Related smells:** Unutilized abstraction [123].
Speculative generality [38]	This smell occurs where an abstraction is created based on speculated requirements. It is often unnecessary that makes things difficult to understand and maintain. **Related smells:** Speculative hierarchy [123]
Lazy class [38]	This smell occurs where a class is not doing enough *i.e.,* it does not realize a concrete responsibility. **Related smells:** Unnecessary abstraction [123].
Switch statement [38]	This smell occurs when switch statements that switch on type codes are spread across the software system instead of exploiting polymorphism.

Table 4: Common code smells

Code Smell / References	Description
	Related smells: Unexploited encapsulation [123], Missing hierarchy [123].
Primitive obsession [38]	This smell occurs when primitive data types are used where an abstraction encapsulating the primitives could serve better. **Related smells:** Missing abstraction [123].
Swiss army knife [16]	This smell arises when the designer attempts to provide all possible uses of the class and ends up in an excessively complex class interface. **Related smells:** Multifaceted abstraction [123].

3.1.3. RQ1.3: How are the smells classified?

An appropriate classification is required to better comprehend a long list of smells based on their characteristics. We collected, categorized, inferred, and synthesized the following set of meta-classification of software smells.

- **Effect-based smell classification:** Mäntylä et al. [72] classified smells based on their effects on software development activities. The categories provided by the classification include *bloaters*, *couplers*, and *change preventers*.

- **Principle-based smell classification:** Samarthyam et al. [41] and Suryanarayana et al. [123] classified design smells based on the primary object-oriented design principle that the smells violate. The principle-based classification divided the smells in four categories namely: *abstraction*, *modularization*, *encapsulation*, and *hierarchy* smells.

- **Artifact characteristics-based smell classification:** Wake [134] proposed a smell classification based on characteristics of the types. Categories such as *data*, *interfaces*, *responsibility*, and *unnecessary complexity* include in his classification. Similarly, Karwin [52] classified SQL antipatterns in the following categories — *logical database design*, *physical database design*, *query*, and *application development* antipatterns.

- **Granularity-based smell classification:** Moha et al. [79] classified smells using two-level classification. At first, a smell is classified in

either *inter-class* and *intra-class* category. The second level of classification assigns non-orthogonal categories *i.e., structural*, *lexical*, and *measurable* to the smells. Similarly, Brown et al. [16] discussed antipatterns classified in three major categories — *software development*, *software architecture*, and *software project management* antipatterns.

Kenneth Bailey [10] discusses a few desirable properties of a classification. By applying them in the context of our study, we propose that an ideal classification of smells must exhibit the following properties.

- **Exhaustive:** classify all the considered smells,
- **Simple:** classify smells within the scope and granularity effortlessly,
- **Consistent:** produce a consistent classification even if it carried out by different people, and
- **Coherent:** produce clearly distinguishable categories without overlaps.

3.1.4. Are smells and antipatterns considered synonyms?

Software engineering researchers and practitioners often use the terms "antipattern" and "smell" interchangeably. Specifically, authors such as Palma et al. [93], Palomba et al. [98], and Linares et al. [65] use both the terms as synonyms. For instance, Linares et al. [65] asserts this notion explicitly — *"...we use the word smells to refer to both code smells and antipatterns, ..."*

Some authors treat antipatterns and smells as quality defects at different granularity. For example, Moha et al. [80] and Moha et al. [79] defined design defects as antipatterns at design granularity and as smells at implementation granularity.

Andrew Koenig [60] coined the term "antipatterns" in 1995 and defined it as follows: *"An antipattern is just like pattern, except that instead of solution it gives something that looks superficially like a solution, but isn't one"*. Hallal et al. [48] also describes antipatterns in this vein — *"something that looks like a good idea, but which backfires badly when applied"*. Based on Andrew's definition, our following interpretation makes antipatterns fundamentally different from smells — *antipatterns get chosen but smells occur, mostly inadvertently.* An antipattern is chosen in the assumption that the

usage will bring more benefits than liabilities whereas smells get introduced due to the lack of knowledge and awareness most of the times.

Brown et al. [16] specify one key characteristic of antipatterns as *"... that generates decidedly negative consequences."* This characteristic makes antipatterns significantly different from smells — a smell is considered as an *indicator* (refer Section 3.1.1) of a problem (rather than the problem itself) whereas antipatterns bring decidedly negative consequences.

An antipattern may lead to smells. For instance, a variant of Singleton introduces sub-type knowledge in a base class leading to cyclic hierarchy [123] smell in the code [35]. Further, the presence of smells may indicate that a certain practice is an antipattern rather than a recommended practice in a given context. For example, the Singleton pattern makes an abstraction difficult to test and hence introduces test smells; the presence of test smells helps us identify that the employed pattern is deteriorating the quality more than helping us solving a design problem.

3.1.5. Implications

We can draw the following implications from the above-discussed research question.

- We found that smells may occur in various stages of software development and impair many dimensions of software quality of different artifact types. This implies that software developers should adopt practices to avoid smells at different granularities, artifacts, and quality dimensions at all stages of software development.

- We identified the core characteristics of software smells. This can help the research community to identify smells even when they are not tagged as smells. For example, it is a recommended practice to avoid accessing external dependencies, such as a database, in a unit test [13]. A non-adherence to the recommended practice exhibits the smell characteristics *violates best practices* and *impacts quality* (maintainability and performance). Therefore, such a violation of the recommended practice could be identified as a test smell despite not being referred to as a smell.

- We elaborated the distinction between antipatterns and smells. This distinction can be further explored in future research on these topics.

3.2. RQ2: How do smells get introduced in software systems?

Authors have explored factors that introduce smells in software systems. We classify such causal factors into the following consolidated list.

- **C1: Lack of skill or awareness** A major reason that cause smells in software systems is poor technical skills of developers and lack of awareness towards writing high quality code. Many authors [123, 77, 22, 125] have pointed out this cause in their studies.
- **C2: Frequently changing requirements** Certain design decisions are made to fulfil the requirements at hand; however, frequent changes in requirements impair the effective decision making and introduce smells [77, 63].
- **C3: Language, platform, or technology constraints** The current literature [121, 77, 59, 63, 22] shows that the chosen technology influences design decisions and could be another reason that leads to smells.
- **C4: Knowledge gap** Missing or complex documentation introduces a knowledge gap which in turn could lead to smells in a software [63, 77].
- **C5: Processes** The adopted processes may help avoid smells to occur or remain in a software system. Therefore, an ineffective or a missing set of processes could also become a cause for software smells [125, 124].
- **C6: Schedule pressure** Developers adopt a quick fix rather than an appropriate solution in the scarcity of time. These quick fixes are a source of smells in software systems [63, 77, 123].
- **C7: Priority to features over quality** Managers consistently pressurise the development teams to deliver new features quickly and ignore the quality of the system [77].
- **C8: Politics** Organizational politics for control, position, and power influence the software quality [22, 63, 122].
- **C9: Team culture** Many authors [3, 22, 125] have recognized the practices and the culture prevailed within a team or an organization as a cause of software smells.

- **C10: Poor human resource planning** Poor planning of human resources required for a software project may force the present development team to adopt quick fixes to meet the deadlines [63].

A cause-based classification can help us understand the categories of factors that causes smells. We propose an alternative to cause-based classification in the form of *actor-based classification*. The actor-based classification assigns the responsibility of the causes to specific actor(s). The identified actors should either correct smells in the current project or learn from the experience so as to avoid these smells in the future. For example, in the current context, we consider three actors — *manager* (representing individuals in the management hierarchy), *technical lead* (the person leading the technical efforts of a software development team), and a software *developer*. Table 5 presents the classification of causes following the actor-based classification scheme. Such a classification can help us in identifying the actionable tasks. For example, if the skill or awareness of software developers is lacking, the actor-based classification suggests that developers as well as their technical-lead are responsible to take a corrective action. Similarly, if appropriate processes are not in-place, it is the responsibility of the technical-lead to deploy them.

Table 5: Actor-based Classification of Smells Causes

Actor\Causes	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**	**C10**
Manager						✓	✓	✓	✓	✓
Technical lead	✓	✓	✓	✓	✓				✓	
Developer	✓			✓					✓	

The above discussed roles and responsibility assignment is an indicative example. The classification has to be adapted based on the team dynamics and the context. For instance, the roles could differ in software development teams that follow different development methods (e.g. agile, traditional waterfall, and hybrid). Furthermore, some development teams are mature to take collective decisions whereas some teams have roles such as scrum master to take decisions that impact the whole team.

3.2.1. Implications

The above exploration consolidates factors reported in the literature that cause smells. It would be interesting to observe their comparative degree of

impact on software smells. Further, we propose a classification that identifies the actors responsible to correct or avoid the causes of specific smells. This explicit identification of responsible actors is actionable; software development teams can improve code quality by making the actors accountable and working with them to correct the underlying factors that lead to specific smells

3.3. RQ3: How do smells affect the software development processes, artifacts, or people?

Smells not only impact software product but also the processes and people working on the product. Table 6 summarizes the impact of smells on software product, process, and people.

Table 6: Impact of Smells

Entity	**Attribute**	**References**
Software product	Maintainability	[11], [100], [82], [137], [136], [138], [119]
	Effort/Cost	[117], [120], [105], [68]
	Reliability	[51], [47], [139], [12], [81], [56]
	Change proneness	[88], [56], [139], [55]
	Testability	[105]
	Performance	[18], [50], [114]
Software development Processes		[115]
People	Morale and motivation	[125], [123]
	Productivity	[125]

Smells have multi-fold impact on the artifacts produced in the software development process and associated quality. Specifically, smells impact maintainability, reliability, testability, performance, and change-proneness of the software. Further, smells also increase effort (and hence cost) required to produce a software.

Presence of excessive amount of smells in a product may influence the outcome of a process; for instance, a high number of smells in a piece of code may lead to pull request rejection [115].

A high number of smells (and hence high technical debt) negatively impact the morale and motivation of the development team and may lead to high attrition [125, 123].

3.3.1. Implications

The above exploration reveals that impact of smells on certain aspects has not been studied in detail. For example, the impact of smells on testability of a software system and productivity of a software development team have been studied only by one study each. Further research in this area can quantify the degree of the smells' impact on diverse product and process quality aspects along with the corresponding implications.

3.4. RQ4: How do smells get detected?

A large body of knowledge exists to detect software smells. Smells have been detected in many studies by employing various techniques. We classify the smell detection strategies in five broad categories; we describe these categories below. Figure 2 shows an abstract layered process flow that we have synthesized by analyzing existing approaches to detect smells using the five categories of smell detection.

1. **Metrics-based smell detection:** A typical metrics-based smell detection method takes source code as the input, prepares a source code model (such as an AST (Abstract Syntax Tree)) (step 1.1 in the figure 2) typically by using a third-party library, detects a set of source code metrics (step 1.2) that capture the characteristics of a set of smells, and detects smells (step 1.3) by applying a suitable threshold [75].
   For example, an instance of the god class smell can be detected using the following set of metrics: WMC (Weighted Methods per Class), ATFD (Access To Foreign Data), and TCC (Tight Class Cohesion) [74, 133]. These metrics are compared against pre-defined thresholds and combined using logical operators. Apart from these, the community frequently uses other metrics such as NOC (Number of Children), NOM (Number of Methods), CBO (Coupling Between Objects), RFC (Response For Class), and LCOM (Lack of Cohesion of Methods) [19] to detect other smells.
2. **Rules/Heuristic-based smell detection:** Smell detection methods that define rules or heuristics [79] (step 2.2 in the figure 2) typically takes source code model (step 2.1) and sometimes additional software

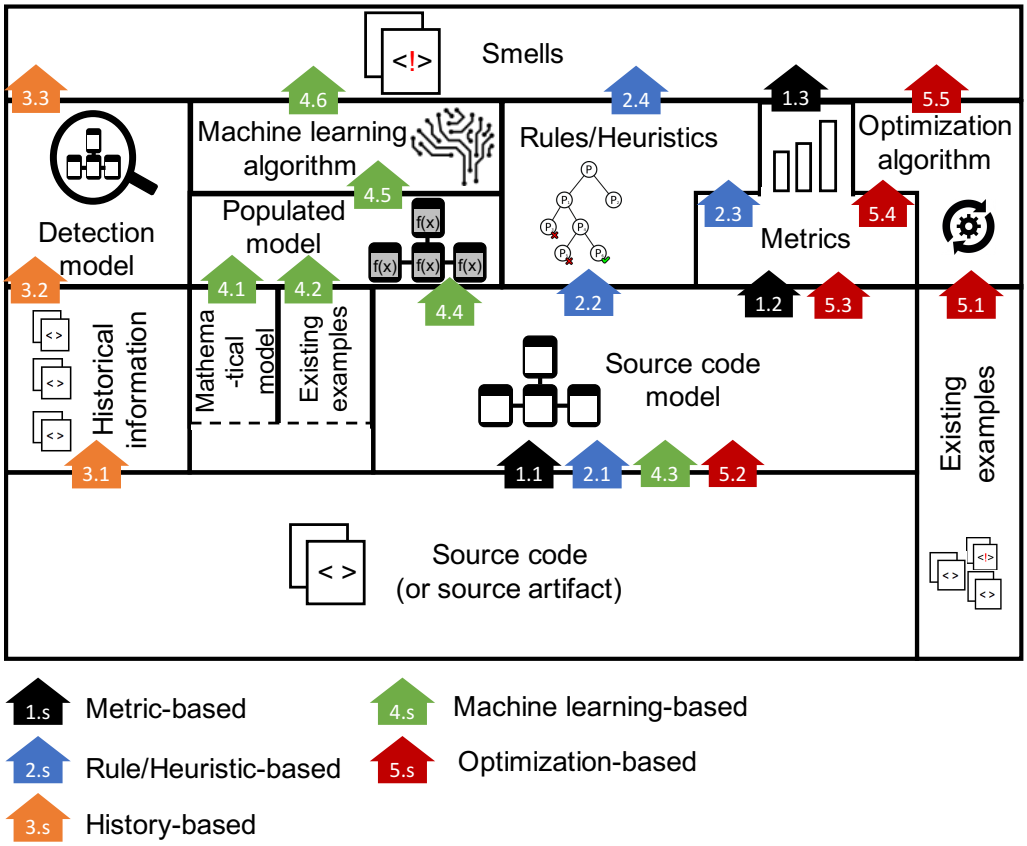

Figure 2: A layered overview of smell detection methods. Each detection method starts from the code (or source artifact) and goes through various steps to detect smells. The direction of the arrows shows the flow direction and annotations on the arrows show the detection method (first part) and the step number (second part).

metrics (step 2.3) as inputs. They detect a set of smells when the defined rules/heuristics get satisfied.

There are many smells that cannot be detected by the currently available metrics alone. For example, we cannot detect rebellious hierarchy, missing abstraction, cyclic hierarchy, and empty catch block smells using commonly used metrics. In such cases, rules or heuristics can be used to detect smells. For example, the cyclic hierarchy [123] smell (when the supertype has knowledge about its subtypes) is detected by defining a rule that checks whether a class is referring to its subclasses. Often, rules or heuristics are combined with metrics to detect smells.

3. **History-based smell detection:** Some authors have detected smells by using source code evolution information [95]. Such methods extract structural information of the code and how it has changed over a period of time (step 3.1 in the figure 2). This information is used by a detection model (step 3.2) to infer smells in the code. For example, by applying association rule mining on a set of methods that have been changed and committed often to the version control system together, divergent change smell can be detected [95].
4. **Machine learning-based smell detection:** Various machine learning methods such as Support Vector Machines [70], and Bayesian Belief Networks [57] have been used to detect smells. A typical machine learning method starts with a mathematical model representing the smell detection problem (step 4.1 in the figure 2). Existing examples (step 4.2) and source code model (step 4.3 and 4.4) could be used to instantiate a concrete populated model. The method results in a set of detected smells by applying a chosen machine learning algorithm (step 4.5) on the populated model. For instance, a Support Vector Machine classifier could be trained using object-oriented metrics attributes for each class. Then the classifier can be used on other programs along with corresponding metrics data to detect smells [70].
5. **Optimization-based smell detection:** Approaches in this category apply optimization algorithms such as genetic algorithms [90] to detect smells. Such methods apply an optimization algorithm on computed software metrics (step 5.4 in the figure 2) and, in some cases, existing examples (step 5.1) of smells to detect new smells in the source code.

Among the surveyed papers, we selected all the papers that employ a smell detection mechanism. We classify these attempts based on the employed smell detection method. Table 7 shows existing attempts to identify smells using one of the smell detection methods. The table also shows number of smells detected by each of the method and target language/artifact.

Each detection method comes with a set of strengths and weaknesses. Metrics-based smell detection is convenient and relatively easy to implement; however, as discussed before, one cannot detect many smells using only commonly known metrics. Another important criticism of metrics-based methods is their dependence on choosing an appropriate set of thresholds, which is a non-trivial challenge. Rule/Heuristic-based detection methods expand the horizon of metrics-based detection by strengthening them with the power

of heuristics defined on source code entities. Therefore, rule/heuristic-based methods combined with metrics offer detection mechanisms that can reveal a high proportion of known smells. History-based methods have a limited applicability because only a few smells are associated with evolutionary changes. Therefore, a source code entity (say a method or a class) that has not necessarily evolved in a certain way to suffer from a smell cannot be detected by history-based methods. Machine learning approaches depend heavily on training data and the lack of such training datasets is a concern [57]. Also, it is still unknown whether machine learning-based detection algorithms can scale to the large number of known smells. Further, optimization-based smell detection methods depend on metric data and corresponding thresholds. This fact makes them suffer from limitations similar to metrics-based methods.

Table 7: Smell Detection Methods and Corresponding References

Smell detection method	**Reference**	**#Smells**	**Languages/ Artifacts**
Metrics-based	[27]	1	Java
	[75]	10	Java, C++
	[83]	2	Java
	[107]	5	Java
	[130]	2	Java
	[85]	1	Java
	[89]	1	Java
	[69]	11	Java
	[37]	5	UML Diagrams
	[14]	7	Aspects-oriented systems
	[113]	3	Java
	[30]	13	JavaScript
	[133]	10	Java
	[91]	2	Java
	[2]	3	NA
	[31]	1	C
	[87]	1	Java

Table 7: Smell Detection Methods and Corresponding References

Smell detection method	**Reference**	**#Smells**	**Languages/ Artifacts**
	[132]	10	JavaScript
	[88]	2	Java
Machine learning-based	[57]	1	Java
	[17]	1	Java
	[58]	3	Java
	[70]	4	Java
	[23]	1	Java
	[71]	NA	Java
History-based	[40]	3	Java
	[95]	5	Java
Rule/Heuristics-based	[102]	5	C
	[28]	8	Use-case Model
	[1]	8	C++
	[34]	1	Java
	[127]	1	Java
	[128]	1	Java
	[21]	4	UML Models
	[7]	1	UML Models
	[99]	5	Java
	[64]	1	Java
	[92]	8	REST APIs
	[79]	4	Java
	[126]	6	Palladio Component Model
	[9]	17	Java
	[110]	30	C#
Optimization-based	[53]	8	Java
	[106]	7	Java

Table 7: Smell Detection Methods and Corresponding References

Smell detection method	Reference	#Smells	Languages/ Artifacts
	[43]	3	Java
	[90]	5	XML (WSDL)

3.4.1. Implications

We identify five categories of smell detection mechanisms. An implication of the categorization for the research community is the positioning of new smell detection methods; the authors can classify their new methods as one of these categories or propose a new smell detection method category.

Among the five types of smell detection methods, metrics-based tools are most popular and relatively easier to develop. On the other hand, researchers are attracted towards machine learning-based methods to overcome the shortcomings of other smell detection methods such as the dependence on choosing appropriate threshold values for metrics. The availability of a standard training dataset would encourage researchers to develop better smell detection tools using machine learning approaches.

3.5. RQ5: What are the open research questions?

Despite the availability of huge amount of literature to understand smells and associated aspects, we perceive many opportunities to expand the domain knowledge.

1 **False-positives and lack of context:** Results produced by the present set of smell detection tools are prone to false-positive instances [35, 58].

- The major reason of the false-positive proneness of the smell detection methods is that metrics and rule-based methods depend heavily on the metrics thresholds. The software engineering community has identified threshold selection as a challenge [53], [37]. There have been many attempts to identify optimal thresholds [36, 66, 33]; however, the proneness to false-positives cannot be eliminated in metrics and rule-based methods since one set of thresholds (or a method to derive thresholds) do not hold good in another context.

- Many authors have asserted that smell detection is a subjective process [73, 96, 84]. As Gil et al. [44] say — *"Bluntly, the code metric values, when inspected out of context, mean nothing."* Similarly, Fontana et al. [35] list a set of commonly detected smells that solve a specific design problem in the real-world.

  We suggest that the identified smells using tools must go through an expert-based scrutiny to finally tag them as quality problems. Essentially, the present set of smell detection methods are not designed to take *context* into account. One potential reason is that it is not easy to define, specify, and capture context. This presents an interesting yet challenging opportunity to significantly improve the *relevance* of detected smells.

- Another interesting concern related to smells in the context of false-positives is that smells are *indicative* by definition and thus it is unfair to tag smells as false-positive based on the context. As shown in Figure 3, a recorded smell could be a false-positive instance (and thus not an actual smell) when it does not fulfil the criteria of a smell by the definition of a smell. When the recorded smell is not a false-positive instance, it could either be a smell which is not a quality problem considering the context of the detected smell. Finally, a detected smell could be a definite quality problem contributing to technical debt. This brings up an interesting insight that researchers and practitioners need to perceive smells (as indicators) differently from definite quality problems.

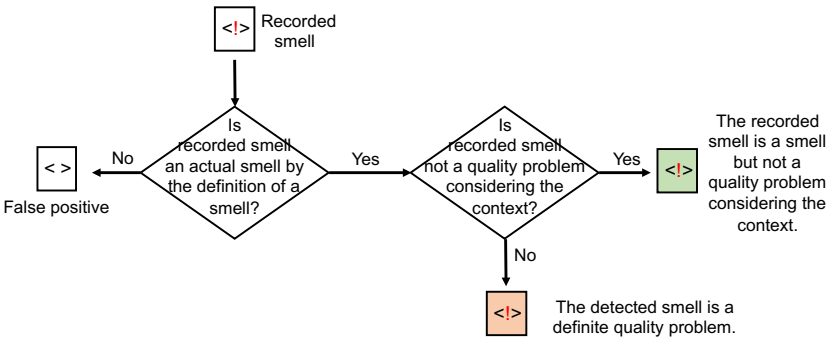

Figure 3: A recorded smell could be a false-positive instance, a smell that is not a quality problem, or a definite quality problem.

For example, consider a tool reports an instance of data class smell in a software system. As explained in Table 4, this smell occurs when a class contains only data fields without any methods. A common practice is to tag the instance of a data class as a false-positive when it is serving a specific purpose in that context [35]. However, we argue that rather than tagging the instance as a false-positive (based on the context), we define smells as being separate from the definite quality problems. A fowl smell in a restaurant may indicate something is rotten, but can also accompany the serving of a strongly smelling cheese.

In a manual inspection, if we find that the class has one method apart from data fields then the reported smell is a false-positive instance since it does not fulfil the condition of a data class smell. On the other hand, if the class only contains data fields without any method definition, it is a smell. As a developer, if one considers the context of the class and infers that the class is being used, for instance, as a DTO (Data Transfer Object) [39] the smell is not a quality problem because it is the result of a conscious design decision. However, if the above case doesn't apply and the developer is using another class (typically a *manager* or a *controller* class) to access and manipulate the data members of the data class, the identified smell is a definite quality problem.

2 Limited detection support for known smells: Table 8 shows all the smell detection tools selected in this study and their corresponding supported smells. It is evident that most of the existing tools support detection of a significantly smaller subset of known smells. Researchers [97, 103, 110] have identified the limited support present for identifying smells in the existing literature. The deficiency poses a serious threat to empirical studies that base their research on a severely low number of smells.

Table 8: Smell Detection Methods and supported smells

References	Detection Method	God class	Feature envy	Shotgun surgery	Data class	Long method	Functional decomposition	Refused bequest	Spaghetti code	Divergent Change	Long Parameter List	Other smells	**Total smells**
[75]	Metrics-based	✓	✓	✓	✓	✓		✓				4	**10**
[83]	Metrics-based											2	**2**
[107]	Metrics-based	✓	✓	✓		✓		✓				0	**5**
[130]	Metrics-based											2	**2**
[34]	Rule/Heuristic-based		✓									0	**1**
[127]	Rule/Heuristic-based											1	**1**
[57]	Machine learning-based	✓										0	**1**
[28]	Rule/Heuristic-based											8	**8**
[88]	Metrics-based	✓	✓									0	**2**
[17]	Machine learning-based					✓						0	**1**
[79]	Rule/Heuristic-based	✓					✓		✓			1	**4**
[85]	Metrics-based		✓									0	**1**
[102]	Rule/Heuristic-based	✓								✓		3	**5**
[89]	Metrics-based	✓										0	**1**
[69]	Metrics-based											11	**11**

Table 8: Smell Detection Methods and supported smells

References	Detection Method	God class	Feature envy	Shotgun surgery	Data class	Long method	Functional decomposition	Refused bequest	Spaghetti code	Divergent Change	Long Parameter List	Other smells	**Total smells**
[21]	Rule/Heuristic-based											4	**4**
[128]	Rule/Heuristic-based					✓						0	**1**
[58]	Machine learning-based	✓					✓		✓			0	**3**
[14]	Metrics-based											7	**7**
[37]	Metrics-based	✓			✓		✓					2	**5**
[126]	Rule/Heuristic-based											6	**6**
[1]	Rule/Heuristic-based											8	**8**
[70]	Machine learning-based	✓					✓		✓			1	**4**
[30]	Metrics-based					✓		✓			✓	10	**13**
[113]	Metrics-based	✓										2	**3**
[9]	Rule/Heuristic-based											17	**17**
[27]	Metrics-based	✓										0	**1**
[64]	Rule/Heuristic-based							✓				0	**1**
[133]	Metrics-based	✓	✓	✓	✓			✓				5	**10**
[106]	Optimization-based	✓	✓		✓		✓		✓		✓	1	**7**

Table 8: Smell Detection Methods and supported smells

References	Detection Method	God class	Feature envy	Shotgun surgery	Data class	Long method	Functional decomposition	Refused bequest	Spaghetti code	Divergent Change	Long Parameter List	Other smells	**Total smells**
[91]	Metrics-based	✓		✓						✓		0	**3**
[53]	Optimization-based	✓	✓	✓	✓		✓		✓		✓	0	**7**
[92]	Rule/Heuristic-based											8	**8**
[2]	Metrics-based	✓		✓		✓						0	**3**
[43]	Optimization-based	✓			✓		✓					0	**3**
[23]	Machine learning-based	✓										0	**1**
[95]	History-based	✓	✓	✓						✓		1	**5**
[31]	Metrics-based					✓						0	**1**
[40]	History-based			✓						✓		1	**3**
[90]	Optimization-based											5	**5**
[87]	Metrics-based		✓									0	**1**
[7]	Rule/Heuristic-based	✓										0	**1**
[99]	Rule/Heuristic-based	✓	✓			✓						2	**5**
[132]	Metrics-based	✓	✓	✓	✓			✓				5	**10**
[110]	Rule/Heuristic-based	✓	✓		✓	✓		✓		✓	✓	23	**30**

Table 8: Smell Detection Methods and supported smells

References	Detection Method	God class	Feature envy	Shotgun surgery	Data class	Long method	Functional decomposition	Refused bequest	Spaghetti code	Divergent Change	Long Parameter List	Other smells	**Total smells**
[71]	Machine learning-based	✓	✓		✓		✓		✓			0	**5**

Figure 4 shows number of studies detecting a specific smell sorted by the number of studies detecting the smells (the top 20 most frequently detected smells). The figure shows that god class smell has been detected the most in the smells literature. On the other hand, some of the smells have been detected only by one study; these smells include parallel inheritance hierarchy [95], closure smells [30], ISP violation [75], hub-like modularization [110], and cyclic hierarchy [110]. Obviously, there are many other smells that have not been detected by any study. The importance and relevance of a smell cannot be determined by its popularity. Hence, the research community also needs to explore the relatively less commonly detected smells and strengthen the quality analysis.

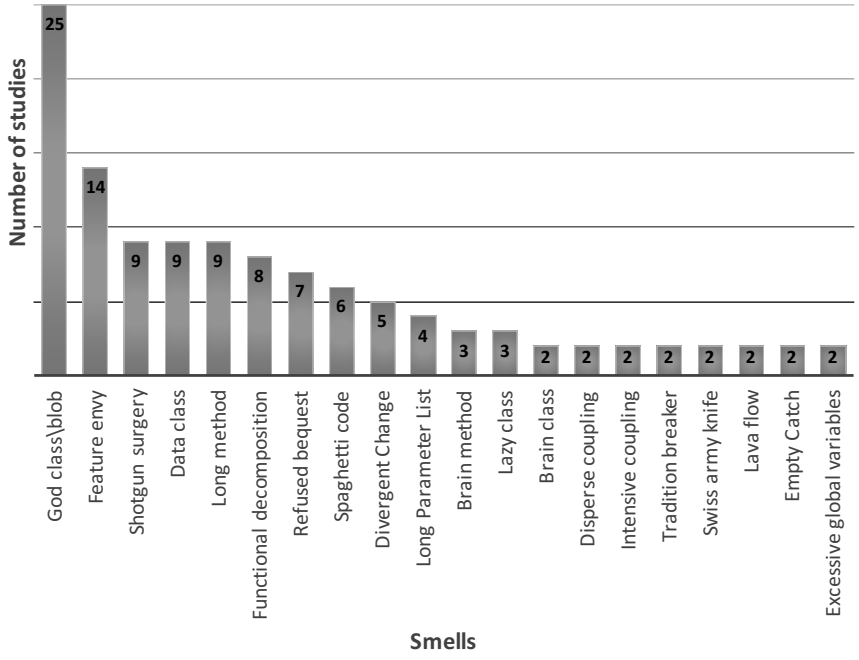

Figure 4: The number of studies detecting a specific smell

Further, academic researchers have concentrated heavily on a single programming language, namely Java [103]. The 46 smell detection methods for source code shown in Table 8 have their targets distributed as follows: 31 for Java, six for models, two for C, two for C++, two for JavaScript, one for C#, and one each for XML and REST APIs.

Expanding the smell detection tools to support a wide range of known smells and diverse programming languages and platforms is another open opportunity.

3 Inconsistent smell definitions and detection methods: The abundance of the smell literature has produced inconsistencies in the definition of smells and their detection methods. For example, god class is one of the most commonly researched smells; however, researchers have defined it differently. Riel et al. [104] has defined it as the class that tend to centralize the knowledge in the system. On the other hand, Gabriela et al. [23] defined it as a class that has too many methods and Mazeiar et al. [107] specified it as the class which is used more extensively than others.

Similarly, based on their description and interpretation, their detection methods also differ significantly and they detect smells inconsistently. Furthermore, in some cases, identical interpretation of smells may also produce different results due to the variation in chosen thresholds of employed metrics.

Even further, metrics tools show inconsistent results even for well-known metrics such as LCOM, CC, and LOC. For example, one tool might implement one variation of LCOM and another tool may realize another or custom variation of the metric while both the tools refer the metric with the same name. Such inconsistencies in smell definition and their detection methods have been identified by the community [103, 8, 41].

It is, therefore, important and relevant to establish a standard with respect to smell definition, their implementation, as well as commonly used metrics.

4 Impact of smells on productivity: In Section 3.3, we present the available literature that discusses the impact of smells on software quality as well as processes and people. It is believed that smells affect mainly maintainability and poor maintainability in turn impacts productivity of the development team. As shown in Section 3.3, the current literature draws connection between impact of smells and maintainability. However, the impact of smells on productivity is not yet explored to a sufficient detail. Other researchers [140] have also identified the need

to better understand the impact of smells. We believe that establishing an explicit and concrete relation between smells and productivity will enhance the adoption of the concepts concerning smells among practitioners.

3.5.1. Implications

In the above discussion, we elaborated on the inherent deficiencies in the present set of smell detection methods. These deficiencies include lack of context and a small number of detectable smells on a very small number of platforms. This analysis clearly calls for effective and widely-applicable smell detection tools and techniques. Inconsistent smell definitions and detection methods indicate the need to set up a standard for smell definitions as well as a verified dataset of smells.

4. Conclusions

This survey presents a synthesized and consolidated overview of the current knowledge in the domain of software smells. We extensively searched a wide range of conferences and journals for the relevant studies published from year 1999 to 2016. The studies selected in all the phases of the selection, an exhaustive smell catalog, as well as the program that generates the smell catalog are made available online.[2]

Our study has explored and identified the following dimensions concerning software smells in the literature.

- We reveal five defining characteristics of software smells: *indicator*, *poor solution*, *violates best practices*, *impacts quality*, and *recurrence*.
- We identify and catalog a wide range of smells (close to 200 at the time of writing this paper) that we made available online and classify them based on 14 focus areas.
- We classify existing smells classifications into four categories: *effect-based*, *principle-based*, *artifact characteristic-based*, and *granularity-based*.
- We curate ten factors that cause smells to occur in a software system. We also classify these causes based on their actors.

[2] `https://github.com/tushartushar/smells`, `http://www.tusharma.in/smells/`

- We categorize existing smell detection methods into five groups: *metrics-based*, *rules/heuristic-based*, *history-based*, *machine learning-based*, and *optimization-based*.

In addition to this, we identify the following gaps and research opportunities in the present set of tools and techniques.

- Existing literature does not differentiate between a smell (as an indicator) and a definite quality problem.
- The community believes that the existing smell detection methods suffer from high false-positive rates. Also, existing methods cannot define, specify, and capture the context of a smell.
- The currently available tools can detect only a very small number of smells. Further, most of the tools largely only support the Java programming language.
- Existing literature has produced inconsistent smell definitions. Similarly, smell detection methods and the corresponding produced results are highly inconsistent.
- The current literature does not establish an explicit connection between smells and their impact on productivity of a software development team.

5. Acknowledgements

The authors would like to thank Vasiliki Efstathiou, Marios Fragkoulis, Stefanos Georgiou, and Theodore Stassinopoulos from the Athens University of Economics and Business for reviewing the early version of the paper and providing useful improvement suggestions. We also would like to convey our sincere thanks to the reviewers of the paper who provided very insightful suggestions to improve the paper.

This work is partially funded by the SENECA project, which is part of the Marie Skłodowska-Curie Innovative Training Networks (ITN-EID). Grant agreement number 642954.

References

[1] Abebe, S. L., Haiduc, S., Tonella, P., Marcus, A., Nov. 2011. The effect of lexicon bad smells on concept location in source code. In: Proceedings - 11th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2011. Fondazione Bruno Kessler, Trento, Italy, IEEE, pp. 125–134.

[2] Abílio, R., Padilha, J., Figueiredo, E., Costa, H., Apr. 2015. Detecting Code Smells in Software Product Lines – An Exploratory Study. In: ITNG '15: Proceedings of the 2015 12th International Conference on Information Technology - New Generations. IEEE Computer Society, pp. 433–438.

[3] Acuña, S. T., Gómez, M., Juristo, N., Aug. 2008. Towards understanding the relationship between team climate and software quality–a quasi-experimental study. Empirical Software Engineering 13 (4), 339–342.

[4] Al Dallal, J., Jan. 2015. Identifying refactoring opportunities in object-oriented code: A systematic literature review. Information and Software Technology 58, 231–249.

[5] Almeida, D., Campos, J. C., Saraiva, J., Silva, J. C., Apr. 2015. Towards a catalog of usability smells. In: SAC '15: Proceedings of the 30th Annual ACM Symposium on Applied Computing. University of Minho, ACM, pp. 175–181.

[6] Alves, P., Figueiredo, E., Ferrari, F., 2014. Avoiding Code Pitfalls in Aspect-Oriented Programming. In: Computational Science and Its Applications – ICCSA 2012. Springer International Publishing, pp. 31–46.

[7] Arcelli, D., Berardinelli, L., Trubiani, C., Jan. 2015. Performance Antipattern Detection through fUML Model Library. In: WOSP '15: Proceedings of the 2015 Workshop on Challenges in Performance Methods for Software Development. University of L'Aquila, ACM, pp. 23–28.

[8] Arcelli Fontana, F., Braione, P., Zanoni, M., 2012. Automatic detection of bad smells in code: An experimental assessment. The Journal of Object Technology 11 (2), 5:1–38.

[9] Arnaoudova, V., Di Penta, M., Antoniol, G., Guéhéneuc, Y.-G., Mar. 2013. A New Family of Software Anti-patterns: Linguistic Anti-patterns. In: CSMR '13: Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering. IEEE Computer Society, pp. 187–196.

[10] Bailey, K. D., 1994. Typologies and taxonomies: an introduction to classification techniques. Vol. 102. Sage.

[11] Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., Binkley, D., Dec. 2012. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: IEEE International Conference on Software Maintenance, ICSM. Universita di Salerno, Salerno, Italy, IEEE, pp. 56–65.

[12] Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., Binkley, D., May 2014. Are test smells really harmful? An empirical study. Empirical Software Engineering 20 (4), 1052–1094.

[13] Beck, K., 2002. Test Driven Development: By Example, 1st Edition. Addison-Wesley Professional.

[14] Bertran, I. M., Garcia, A., von Staa, A., Mar. 2011. An exploratory study of code smells in evolving aspect-oriented systems. In: AOSD '11: Proceedings of the tenth international conference on Aspect-oriented software development. Pontifical Catholic University of Rio de Janeiro, ACM, p. 203.

[15] Binkley, D., Gold, N., Harman, M., Li, Z., Mahdavi, K., Wegener, J., Dec. 2008. Dependence Anti Patterns. In: Aramis 2008 - 1st International Workshop on Automated engineeRing of Autonomous and runtiMe evolvIng Systems, and ASE2008 the 23rd IEEE/ACM Int. Conf. Automated Software Engineering. King's College London, London, United Kingdom, IEEE, pp. 25–34.

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

[17] Bryton, S., Brito E Abreu, F., Monteiro, M., Dec. 2010. Reducing subjectivity in code smells detection: Experimenting with the Long

Method. In: Proceedings - 7th International Conference on the Quality of Information and Communications Technology, QUATIC 2010. Faculdade de Ciencias e Tecnologia, New University of Lisbon, Caparica, Portugal, IEEE, pp. 337–342.

[18] Chen, T.-H., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., Flora, P., May 2014. Detecting performance anti-patterns for applications developed using object-relational mapping. In: ICSE 2014: Proceedings of the 36th International Conference on Software Engineering. Queen's University, Kingston, ACM, pp. 1001–1012.

[19] Chidamber, S. R., Kemerer, C. F., Jun. 1994. A metrics suite for object oriented design. IEEE Transaction of Software Engineering 20 (6), 476–493.

[20] Cortellessa, V., Di Marco, A., Trubiani, C., Feb. 2014. An approach for modeling and detecting software performance antipatterns based on first-order logics. Software and Systems Modeling (SoSyM) 13 (1), 391–432.

[21] Cortellessa, V., Martens, A., Reussner, R., Trubiani, C., Apr. 2010. A process to effectively identify "guilty" performance antipatterns. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Universita degli Studi dell'Aquila, L'Aquila, Italy, Springer Berlin Heidelberg, pp. 368–382.

[22] Curcio, K., Malucelli, A., Reinehr, S., Paludo, M. A., Nov. 2016. An analysis of the factors determining software product quality: A comparative study. Computer Standards & Interfaces 48, 10–18.

[23] Czibula, G., Marian, Z., Czibula, I. G., Mar. 2015. Detecting software design defects using relational association rule mining. Knowledge and Information Systems 42 (3), 545–577.

[24] da Silva Sousa, L., May 2016. Spotting design problems with smell agglomerations. In: ICSE '16: Proceedings of the 38th International Conference on Software Engineering Companion. Pontifical Catholic University of Rio de Janeiro, ACM, pp. 863–866.

[25] Das, T. K., Dingel, J., Jul. 2016. Model development guidelines for UML-RT: conventions, patterns and antipatterns. Software & Systems Modeling, 1–36.

[26] Deursen, A. V., Moonen, L., Bergh, A. V. D., Kok, G., 2001. Refactoring test code. In: Marchesi, M. (Ed.), Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001). University of Cagliari, pp. 92–95.

[27] Dexun, J., Peijun, M., Xiaohong, S., Tiantian, W., Sep. 2013. Detection and Refactoring of Bad Smell Caused by Large Scale. International Journal of Software Engineering & Applications 4 (5), 1–13.

[28] El-Attar, M., Miller, J., Feb. 2009. Improving the quality of use case models using antipatterns. Software & Systems Modeling 9 (2), 141–160.

[29] Fard, A. M., Mesbah, A., Jan. 2013. JSNOSE: Detecting javascript code smells. In: IEEE 13th International Working Conference on Source Code Analysis and Manipulation, SCAM 2013. The University of British Columbia, Vancouver, Canada, IEEE, pp. 116–125.

[30] Fard, A. M., Mesbah, A., Jan. 2013. JSNOSE: Detecting javascript code smells. In: IEEE 13th International Working Conference on Source Code Analysis and Manipulation, SCAM 2013. The University of British Columbia, Vancouver, Canada, IEEE, pp. 116–125.

[31] Fenske, W., Schulze, S., Meyer, D., Saake, G., Nov. 2015. When code smells twice as much: Metric-based detection of variability-aware code smells. In: 2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation, SCAM 2015 - Proceedings. Otto von Guericke University of Magdeburg, Magdeburg, Germany, IEEE, pp. 171–180.

[32] Fernandes, E., Oliveira, J., Vale, G., Paiva, T., Figueiredo, E., Jun. 2016. A review-based comparative study of bad smell detection tools. In: EASE '16: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering. Federal University of Minas Gerais, ACM, pp. 18–12.

[33] Ferreira, K. A. M., Bigonha, M. A. S., Bigonha, R. S., Mendes, L. F. O., Almeida, H. C., Feb. 2012. Identifying thresholds for object-oriented software metrics. Journal of Systems and Software 85 (2), 244–257.

[34] Fokaefs, M., Tsantalis, N., Chatzigeorgiou, A., 2007. JDeodorant: Identification and Removal of Feature Envy Bad Smells. In: 2007 IEEE International Conference on Software Maintenance. Panepistimion Makedonias, Thessaloniki, Greece, IEEE, pp. 519–520.

[35] Fontana, F. A., Dietrich, J., Walter, B., Yamashita, A., Zanoni, M., 2016. Antipattern and Code Smell False Positives: Preliminary Conceptualization and Classification. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE, pp. 609–613.

[36] Fontana, F. A., Ferme, V., Zanoni, M., Yamashita, A., May 2015. Automatic metric thresholds derivation for code smell detection. In: WETSoM '15: Proceedings of the Sixth International Workshop on Emerging Trends in Software Metrics. University of Lugano, IEEE Press, pp. 44–53.

[37] Fourati, R., Bouassida, N., Abdallah, H. B., 2011. A Metric-Based Approach for Anti-pattern Detection in UML Designs. In: Computer and Information Science 2011. Springer Berlin Heidelberg, pp. 17–33.

[38] Fowler, M., 1999. Refactoring: Improving the Design of Existing Programs, 1st Edition. Addison-Wesley Professional.

[39] Fowler, M., 2002. Patterns of Enterprise Application Architecture, 1st Edition. Addison-Wesley Professional.

[40] Fu, S., Shen, B., Nov. 2015. Code Bad Smell Detection through Evolutionary Data Mining. In: International Symposium on Empirical Software Engineering and Measurement. Shanghai Jiaotong University, Shanghai, China, IEEE, pp. 41–49.

[41] Ganesh, S., Sharma, T., Suryanarayana, G., Jun. 2013. Towards a principle-based classification of structural design smells. Journal of Object Technology 12 (2), 1:1–29.

[42] Garcia, J., Popescu, D., Edwards, G., Medvidovic, N., 2009. Toward a catalogue of architectural bad smells. In: Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems. QoSA '09. Springer-Verlag, pp. 146–162.

[43] Ghannem, A., El Boussaidi, G., Kessentini, M., Mar. 2015. On the use of design defect examples to detect model refactoring opportunities. Software Quality Journal, 1–19.

[44] Gil, J. Y., Lalouche, G., 2016. When do Software Complexity Metrics Mean Nothing? – When Examined out of Context. The Journal of Object Technology 15 (1), 2:1.

[45] Greiler, M., van Deursen, A., Storey, M.-A., Jan. 2013. Automated Detection of Test Fixture Strategies and Smells. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST). IEEE, pp. 322–331.

[46] Guerrouj, L., Kermansaravi, Z., Arnaoudova, V., Fung, B. C. M., Khomh, F., Antoniol, G., Guéhéneuc, Y.-G., May 2016. Investigating the relation between lexical smells and change- and fault-proneness: an empirical study. Software Quality Journal, 1–30.

[47] Hall, T., Zhang, M., Bowes, D., Sun, Y., Sep. 2014. Some Code Smells Have a Significant but Small Effect on Faults. ACM Transactions on Software Engineering and Methodology (TOSEM) 23 (4), 33–39.

[48] Hallal, H. H., Alikacem, E., Tunney, W. P., Boroday, S., Petrenko, A., Sep. 2004. Antipattern-Based Detection of Deficiencies in Java Multithreaded Software. In: QSIC '04: Proceedings of the Quality Software, Fourth International Conference. Cent de Recherche Informatique de Montreal, IEEE Computer Society, pp. 258–267.

[49] Hauptmann, B., Junker, M., Eder, S., Heinemann, L., Vaas, R., Braun, P., May 2013. Hunting for smells in natural language tests. In: ICSE '13: Proceedings of the 2013 International Conference on Software Engineering. Technical University of Munich, IEEE Press, pp. 1217–1220.

[50] Hecht, G., Moha, N., Rouvoy, R., May 2016. An empirical study of the performance impacts of Android code smells. In: MOBILESoft

'16: Proceedings of the International Workshop on Mobile Software Engineering and Systems. Universite Lille 2 Droit et Sante, ACM.

[51] Jaafar, F., Guéhéneuc, Y.-G., Hamel, S., Khomh, F., 2013. Mining the relationship between anti-patterns dependencies and fault-proneness. In: Proceedings - Working Conference on Reverse Engineering, WCRE. Ecole Polytechnique de Montreal, Montreal, Canada, IEEE, pp. 351–360.

[52] Karwin, B., 2010. SQL Antipatterns: Avoiding the Pitfalls of Database Programming, 1st Edition. Pragmatic Bookshelf.

[53] Kessentini, W., Kessentini, M., Sahraoui, H., Bechikh, S., Ouni, A., 2014. A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection. IEEE Transactions on Software Engineering 40 (9), 841–861.

[54] Khan, Y. A., El-Attar, M., 2016. Using model transformation to refactor use case models based on antipatterns. Information Systems Frontiers 18 (1), 171–204.

[55] Khomh, F., Di Penta, M., Guéhéneuc, Y.-G., Dec. 2009. An Exploratory Study of the Impact of Code Smells on Software Change-proneness. In: 2009 16th Working Conference on Reverse Engineering. Ecole Polytechnique de Montreal, Montreal, Canada, IEEE, pp. 75–84.

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

[57] Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., Sahraoui, H., Aug. 2009. A Bayesian Approach for the Detection of Code and Design Smells. In: QSIC '09: Proceedings of the 2009 Ninth International Conference on Quality Software. IEEE Computer Society, pp. 305–314.

[58] Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., Sahraoui, H., 2011. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. In: Journal of Systems and Software. Ecole Polytechnique de Montreal, Montreal, Canada, pp. 559–572.

[59] Kleinschmager, S., Hanenberg, S., Robbes, R., Stefik, A., 2012. Do static type systems improve the maintainability of software systems? An empirical study. In: 2012 IEEE 20th International Conference on Program Comprehension (ICPC). Universitat Duisburg-Essen, Essen, Germany, IEEE, pp. 153–162.

[60] Koenig, A., 1995. Patterns and antipatterns. JOOP 8 (1), 46–48.

[61] Král, J., Žemlička, M., Dec. 2007. The most important service-oriented antipatterns. In: 2nd International Conference on Software Engineering Advances - ICSEA 2007. Charles University in Prague, Prague, Czech Republic, IEEE, pp. 29–29.

[62] Lauder, A., Kent, S., 2000. Legacy System Anti-Patterns and a Pattern-Oriented Migration Response. In: Systems Engineering for Business Process Change. Springer London, pp. 239–250.

[63] Lavallée, M., Robillard, P. N., Aug. 2015. Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality. In: Proceedings - International Conference on Software Engineering. Polytechnique Montréal, Montreal, Canada, IEEE, pp. 677–687.

[64] Ligu, E., Chatzigeorgiou, A., Chaikalis, T., Ygeionomakis, N., Sept 2013. Identification of refused bequest code smells. In: 2013 IEEE International Conference on Software Maintenance. pp. 392–395.

[65] Linares-Vásquez, M., Klock, S., McMillan, C., Sabané, A., Poshyvanyk, D., Guéhéneuc, Y.-G., Jun. 2014. Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in Java mobile apps. In: ICPC 2014: Proceedings of the 22nd International Conference on Program Comprehension. The College of William and Mary, ACM, pp. 232–243.

[66] Liu, H., Liu, Q., Niu, Z., Liu, Y., Jun. 2016. Dynamic and Automatic Feedback-Based Threshold Adaptation for Code Smell Detection. IEEE Transactions on Software Engineering 42 (6), 544–558.

[67] Long, J., Jul. 2001. Software reuse antipatterns. ACM SIGSOFT Software Engineering Notes 26 (4), 68–76.

[68] MacCormack, A., Sturtevant, D. J., 2016. Technical debt and system architecture: The impact of coupling on defect-related activity. Journal of Systems and Software 120, 170 – 182.

[69] Macia, I., Garcia, A., von Staa, A., Dec. 2010. Defining and applying detection strategies for aspect-oriented code smells. In: Proceedings - 24th Brazilian Symposium on Software Engineering, SBES 2010. Pontificia Universidade Catolica do Rio de Janeiro, Rio de Janeiro, Brazil, IEEE, pp. 60–69.

[70] Maiga, A., Ali, N., Bhattacharya, N., Sabané, A., Guéhéneuc, Y.-G., Antoniol, G., Aïmeur, E., Sep. 2012. Support vector machines for anti-pattern detection. In: ASE 2012: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. Polytechnic School of Montreal, ACM, pp. 278–281.

[71] Mansoor, U., Kessentini, M., Maxim, B. R., Deb, K., Feb. 2016. Multi-objective code-smells detection using good and bad design examples. Software Quality Journal, 1–24.

[72] Mäntylä, M., Vanhanen, J., Lassenius, C., Sep. 2003. A Taxonomy and an Initial Empirical Study of Bad Smells in Code. In: ICSM '03: Proceedings of the International Conference on Software Maintenance. IEEE Computer Society.

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

[74] Marinescu, R., 2004. Detection strategies: Metrics-based rules for detecting design flaws. In: Proceedings of the 20th IEEE International Conference on Software Maintenance. ICSM '04. IEEE Computer Society, pp. 350–359.

[75] Marinescu, R., Dec. 2005. Measurement and quality in object-oriented design. In: 21st IEEE International Conference on Software Maintenance (ICSM'05). Universitatea Politehnica din Timisoara, Timisoara, Romania, IEEE, pp. 701–704.

[76] Marquardt, K., 2001. Dependency structuresarchitectural diagnoses and therapies. In: Proceedings of the EuroPLop.

[77] Martini, A., Bosch, J., Chaudron, M., Aug 2014. Architecture technical debt: Understanding causes and a qualitative model. In: 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications. pp. 85–92.

[78] Mens, T., Tourwé, T., Feb. 2004. A Survey of Software Refactoring. IEEE Transactions on Software Engineering 30 (2), 126–139.

[79] Moha, N., Guéhéneuc, Y., Duchien, L., Meur, A. L., 2010. DECOR: A method for the specification and detection of code and design smells. IEEE Trans. Software Eng. 36 (1), 20–36.

[80] Moha, N., Guéhéneuc, Y.-G., 2007. Decor: a tool for the detection of design defects. In: ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. University of Montreal, ACM, pp. 527–528.

[81] Monden, A., Nakae, D., Kamiya, T., Sato, S.-i., Matsumoto, K.-i., Jun. 2002. Software Quality Analysis by Code Clones in Industrial Legacy Software. In: METRICS '02: Proceedings of the 8th International Symposium on Software Metrics. IEEE Computer Society, p. 87.

[82] Moonen, L., Yamashita, A., Sep. 2012. Do code smells reflect important maintainability aspects? In: ICSM '12: Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM). Simula Research Laboratory, IEEE Computer Society.

[83] Munro, M. J., Sep. 2005. Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. In: METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05). University of Strathclyde, IEEE Computer Society, pp. 15–15.

[84] Murphy-Hill, E., Black, A. P., 2008. Seven habits of a highly effective smell detector. In: the 2008 international workshop. Portland State University, Portland, United States, ACM Press, pp. 36–40.

[85] Murphy-Hill, E., Black, A. P., Oct. 2010. An interactive ambient visualization for code smells. In: SOFTVIS '10: Proceedings of the 5th international symposium on Software visualization. North Carolina State University, ACM.

[86] Nguyen, H. V., Nguyen, H. A., Nguyen, T. T., Nguyen, A. T., Nguyen, T. N., Sep. 2012. Detection of embedded code smells in dynamic web applications. In: ASE 2012: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. Iowa State University, ACM, pp. 282–285.

[87] Nongpong, K., Jan. 2015. Feature envy factor: A metric for automatic feature envy detection. In: Proceedings of the 2015-7th International Conference on Knowledge and Smart Technology, KST 2015. Assumption University, Bangkok, Bangkok, Thailand, IEEE, pp. 7–12.

[88] Olbrich, S., Cruzes, D. S., Basili, V., Zazworka, N., Aug. 2009. The evolution and impact of code smells: A case study of two open source systems. In: 2009 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, pp. 390–400.

[89] Oliveto, R., Khomh, F., Antoniol, G., Guéhéneuc, Y.-G., Mar. 2010. Numerical Signatures of Antipatterns: An Approach Based on B-Splines. In: CSMR '10: Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering. IEEE Computer Society, pp. 248–251.

[90] Ouni, A., Kula, R. G., Kessentini, M., Inoue, K., Jul. 2015. Web Service Antipatterns Detection Using Genetic Programming. In: GECCO '15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation. Osaka University, ACM, pp. 1351–1358.

[91] Padilha, J., Pereira, J., Figueiredo, E., Almeida, J., Garcia, A., Sant'Anna, C., Jan. 2014. On the effectiveness of concern metrics to detect code smells: An empirical study. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Universidade Federal de Minas Gerais, Belo Horizonte, Brazil, Springer International Publishing, pp. 656–671.

[92] Palma, F., Dubois, J., Moha, N., Guéhéneuc, Y.-G., Jan. 2014. Detection of REST patterns and antipatterns: A heuristics-based approach. In: Franch, X., Ghose, A. K., Lewis, G. A., Bhiri, S. (Eds.), Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Universite du

Quebec a Montreal, Montreal, Canada, Springer Berlin Heidelberg, pp. 230–244.

[93] Palma, F., Moha, N., Guéhéneuc, Y.-G., Jan. 2013. Detection of process antipatterns: A BPEL perspective. In: Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC. Eŕcole Polytechnique, Canada, IEEE, pp. 173–177.

[94] Palma, F., Mohay, N., Jan. 2015. A study on the taxonomy of service antipatterns. In: 2015 IEEE 2nd International Workshop on Patterns Promotion and Anti-Patterns Prevention, PPAP 2015 - Proceedings. Ecole Polytechnique de Montreal, Montreal, Canada, IEEE, pp. 5–8.

[95] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., De Lucia, A., May 2015. Mining version histories for detecting code smells. IEEE Transactions on Software Engineering 41 (5), 462–489.

[96] Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., Lucia, A. D., Jul. 2014. Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In: 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 101–110.

[97] Palomba, F., De Lucia, A., Bavota, G., Oliveto, R., 2014. Anti-Pattern Detection. In: Anti-pattern detection: Methods, challenges, and open issues. Elsevier, pp. 201–238.

[98] Palomba, F., Nucci, D. D., Tufano, M., Bavota, G., Oliveto, R., Poshyvanyk, D., De Lucia, A., 2015. Landfill: An open dataset of code smells with public evaluation. In: Proceedings of the 12th Working Conference on Mining Software Repositories. MSR '15. IEEE Press, pp. 482–485.

[99] Palomba, F., Panichella, A., De Lucia, A., Oliveto, R., Zaidman, A., 2016. A textual-based technique for Smell Detection. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC). Universita di Salerno, Salerno, Italy, IEEE, pp. 1–10.

[100] Perepletchikov, M., Ryan, C., Aug. 2011. A controlled experiment for evaluating the impact of coupling on the maintainability of service-oriented software. IEEE Transactions on Software Engineering 37 (4), 449–465.

[101] Peters, R., Zaidman, A., 2012. Evaluating the lifespan of code smells using software repository mining. In: Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering. CSMR '12. IEEE Computer Society, pp. 411–416.

[102] Rama, G. M., Feb. 2010. A desiderata for refactoring-based software modularity improvement. In: ISEC '10: Proceedings of the 3rd India software engineering conference. Infosys Technologies Limited India, ACM, pp. 93–102.

[103] Rasool, G., Arshad, Z., Nov. 2015. A review of code smell mining techniques. Journal of Software: Evolution and Process 27 (11), 867–895.

[104] Riel, A. J., 1996. Object-Oriented Design Heuristics, 1st Edition. Addison-Wesley.

[105] Sabané, A., Di Penta, M., Antoniol, G., Guéhéneuc, Y.-G., Mar. 2013. A Study on the Relation between Antipatterns and the Cost of Class Unit Testing. In: CSMR '13: Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering. IEEE Computer Society, pp. 167–176.

[106] Sahin, D., Kessentini, M., Bechikh, S., Deb, K., Oct. 2014. Code-Smell Detection as a Bilevel Problem. ACM Transactions on Software Engineering and Methodology (TOSEM) 24 (1), 6–44.

[107] Salehie, M., Li, S., Tahvildari, L., Jun. 2006. A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws. In: ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06). University of Waterloo, IEEE Computer Society, pp. 159–168.

[108] Sharma, T., Nov. 2017. Detailed overview of software smells. URL `https://doi.org/10.5281/zenodo.1066043`

[109] Sharma, T., Fragkoulis, M., Spinellis, D., 2016. Does your configuration code smell? In: Proceedings of the 13th International Workshop on Mining Software Repositories. MSR'16. pp. 189–200.

[110] Sharma, T., Mishra, P., Tiwari, R., 2016. Designite — A Software Design Quality Assessment Tool. In: Proceedings of the First International Workshop on Bringing Architecture Design Thinking into Developers' Daily Activities. BRIDGE '16. ACM.

[111] Sharma, T., Spinellis, D., Nov. 2017. Definitions of a software smell. URL https://doi.org/10.5281/zenodo.1066135

[112] Sharma, T., Spinellis, D., Nov. 2017. Selected Resources for a Literature Survey on Software Smells.
URL https://doi.org/10.5281/zenodo.1069330

[113] Sharma, V. S., Anwer, S., Dec. 2013. Detecting Performance Antipatterns before Migrating to the Cloud. In: CLOUDCOM '13: Proceedings of the 2013 IEEE International Conference on Cloud Computing Technology and Science - Volume 01. IEEE Computer Society, pp. 148–151.

[114] Sharma, V. S., Anwer, S., Jan. 2014. Performance antipatterns: Detection and evaluation of their effects in the cloud. In: Proceedings - 2014 IEEE International Conference on Services Computing, SCC 2014. Accenture Services Pvt Ltd., India, Bangalore, India, IEEE, pp. 758–765.

[115] Silva, M. C. O., Valente, M. T., Terra, R., 2016. Does technical debt lead to the rejection of pull requests? CoRR abs/1604.01450.

[116] Singh, S., Kaur, S., 2017. A systematic literature review: Refactoring for disclosing code smells in object oriented software. Ain Shams Engineering Journal.
URL http://www.sciencedirect.com/science/article/pii/S2090447917300412

[117] Sjoberg, D. I. K., Yamashita, A., Anda, B., Mockus, A., Dyba, T., Aug. 2013. Quantifying the Effect of Code Smells on Maintenance Effort. IEEE Transactions on Software Engineering 39 (8), 1144–1156.

[118] Smith, C., Dec. 2000. Software performance antipatterns. In: Proceedings Second International Workshop on Software and Performance WOSP 2000. Performance Engineering Services, Santa Fe, United States, pp. 127–136a.

[119] Soh, Z., Yamashita, A., Khomh, F., Guéhéneuc, Y.-G., 2016. Do Code Smells Impact the Effort of Different Maintenance Programming Activities? In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE, pp. 393–402.

[120] Spínola, R. O., Zazworka, N., Vetrò, A., Seaman, C., Shull, F., 2013. Investigating technical debt folklore: Shedding some light on technical debt opinion. In: Proceedings of the 4th International Workshop on Managing Technical Debt. MTD '13. IEEE Press, pp. 1–7.

[121] Stella, L. F. F., Jarzabek, S., Wadhwa, B., Dec. 2008. A comparative study of maintainability of web applications on J2EE, .NET and ruby on rails. In: Proceedings - 10th IEEE International Symposium on Web Site Evolution, WSE 2008. National University of Singapore, Singapore City, Singapore, IEEE, pp. 93–99.

[122] Stribrny, S., Mackin, F. B., Sep. 2006. When politics overshadow software quality. IEEE Software 23 (5), 72–73.

[123] Suryanarayana, G., Samarthyam, G., Sharma, T., 2014. Refactoring for Software Design Smells: Managing Technical Debt, 1st Edition. Morgan Kaufmann.

[124] Suryanarayana, G., Sharma, T., Samarthyam, G., 2015. Software Process versus Design Quality: Tug of War? IEEE Software 32 (4), 7–11.

[125] Tom, E., Aurum, A., Vidgen, R., 2013. An exploration of technical debt. Journal of Systems and Software 86 (6), 1498 – 1516.

[126] Trubiani, C., Koziolek, A., Mar. 2011. Detection and solution of software performance antipatterns in palladio architectural models. In: ICPE '11: Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering. Karlsruhe Institute of Technology, ACM, pp. 11–11.

[127] Tsantalis, N., Chaikalis, T., Chatzigeorgiou, A., Apr. 2008. JDeodorant: Identification and Removal of Type-Checking Bad Smells. In: CSMR '08: Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering. University of Macedonia, IEEE Computer Society, pp. 329–331.

[128] Tsantalis, N., Chatzigeorgiou, A., Oct. 2011. Identification of extract method refactoring opportunities for the decomposition of methods. Journal of Systems & Software 84 (10), 1757–1782.

[129] Van Emden, E., Moonen, L., 2002. Java quality assurance by detecting code smells. Ninth Working Conference on Reverse Engineering, 97–106.

[130] Van Rompaey, B., Du Bois, B., Demeyer, S., Rieger, M., Dec. 2007. On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test. IEEE Transactions on Software Engineering 33 (12), 800–817.

[131] Vetr, A., Ardito, L., Procaccianti, G., Morisio, M., 2013. Definition, implementation and validation of energy code smells: an exploratory study on an embedded system. ThinkMind, pp. 34–39.

[132] Vidal, S., Vazquez, H., Díaz-Pace, J. A., Marcos, C., Garcia, A., Oizumi, W., Feb. 2016. JSpIRIT: A flexible tool for the analysis of code smells. In: Proceedings - International Conference of the Chilean Computer Science Society, SCCC. Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Argentina, IEEE, pp. 1–6.

[133] Vidal, S. A., Marcos, C., Díaz-Pace, J. A., 2014. An approach to prioritize code smells for refactoring. Automated Software Engineering 23 (3), 501–532.

[134] Wake, W. C., 2003. Refactoring Workbook, 1st Edition. Addison-Wesley Longman Publishing Co., Inc.

[135] Wang, C., Hirasawa, S., Takizawa, H., Kobayashi, H., May 2014. A Platform-Specific Code Smell Alert System for High Performance Computing Applications. In: IPDPSW '14: Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops. IEEE Computer Society, pp. 652–661.

[136] Yamashita, A., 2014. Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data. Empirical Software Engineering 19 (4), 1111–1143.

[137] Yamashita, A., Moonen, L., 2013. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In: Proceedings of the 2013 International Conference on Software Engineering. ICSE '13. IEEE Press, pp. 682–691.

[138] Yamashita, A., Moonen, L., Dec. 2013. To what extent can maintenance problems be predicted by code smell detection? - an empirical study. Information and Software Technology 55 (12), 2223–2242.

[139] Zazworka, N., Shaw, M. A., Shull, F., Seaman, C., May 2011. Investigating the impact of design debt on software quality. In: MTD '11: Proceedings of the 2nd Workshop on Managing Technical Debt. Fraunhofer USA, Inc., ACM, pp. 17–23.

[140] Zhang, M., Hall, T., Baddoo, N., Apr. 2011. Code Bad Smells: A review of current knowledge. Journal of Software Maintenance and Evolution 23 (3), 179–202.