The Journal of Systems and Software 167 (2020) 110610

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Code smells and refactoring: A tertiary systematic review of challenges and observations

Guilherme Lacerda[a,b,*], Fabio Petrillo[c], Marcelo Pimenta[b], Yann Gaël Guéhéneuc[d]

[a] University of Vale do Rio dos Sinos, Polytechnic School São Leopoldo, RS, Brazil
[b] Federal University of Rio Grande do Sul, Institute of Informatics Porto Alegre, RS, Brazil
[c] University of Quebec at Chicoutimi, Department of Computer Science & Mathematics Chicoutimi, Quebec, Canada
[d] Concordia University, Departement of Computer Science and Software Engineering Montreal, Quebec, Canada

ARTICLE INFO

Article history:
Received 11 September 2019
Revised 25 February 2020
Accepted 18 April 2020
Available online 30 April 2020

Keywords:
Code smells
Refactoring
Tertiary systematic review

ABSTRACT

Refactoring and smells have been well researched by the software-engineering research community these past decades. Several secondary studies have been published on code smells, discussing their implications on software quality, their impact on maintenance and evolution, and existing tools for their detection. Other secondary studies addressed refactoring, discussing refactoring techniques, opportunities for refactoring, impact on quality, and tools support.

In this paper, we present a tertiary systematic literature review of previous surveys, secondary systematic literature reviews, and systematic mappings. We identify the main observations (*what we know*) and challenges (*what we do not know*) on code smells and refactoring. We perform this tertiary review using eight scientific databases, based on a set of five research questions, identifying 40 secondary studies between 1992 and 2018.

We organize the main observations and challenges about code smell and their refactoring into: smells definitions, most common code-smell detection approaches, code-smell detection tools, most common refactoring, and refactoring tools. We show that code smells and refactoring have a strong relationship with quality attributes, *i.e.,* with understandability, maintainability, testability, complexity, functionality, and reusability. We argue that code smells and refactoring could be considered as the two faces of a same coin. Besides, we identify how refactoring affects quality attributes, more than code smells. We also discuss the implications of this work for practitioners, researchers, and instructors. We identify 13 open issues that could guide future research work.

Thus, we want to highlight the gap between code smells and refactoring in the current state of software-engineering research. We wish that this work could help the software-engineering research community in collaborating on future work on code smells and refactoring.

1. Introduction

Software maintenance is an essential activity for any software system. According to Welf Löwe and Panas (2005) and Telea and Voinea (2011), 50% to 80% of software costs are related to maintenance activities: repairing design and implementation faults, adapting software to a different environment (hardware, OS), and adding or modifying functionalities. Software maintenance is difficult because of the lack of helpful documentation, large and complex source code becomes the only reliable source of information about a system (Welf Löwe and Panas, 2005).

Several studies provided a broad overview of pitfalls (Webster, 1995), anti-patterns (Brown et al., 1998), and smells (Fowler et al., 1999). Although there are many contexts where smells can be found, like models (Misbhauddin and Alshayeb, 2015), tests (Garousi and Mäntylä, 2016), requirements (Alves et al., 2016), architecture (Besker et al., 2018; Li et al., 2015), and services (Sabir et al., 2018), our main focus is smells found in source code, a.k.a. code smells, because these impact maintainability negatively (Hall et al., 2014).

Code smells are violations of coding design principles (Fowler and Beck, 2019). They increase technical debt (Kruchten et al., 2012), affecting software maintenance (Sjøberg

* Corresponding author.
E-mail addresses: guilhermeslacerda@gmail.com (G. Lacerda), fabio@petrillo.com (F. Petrillo), mpimenta@inf.ufrgs.br (M. Pimenta), yann-gael.gueheneuc@concordia.ca (Y.G. Guéhéneuc).

https://doi.org/10.1016/j.jss.2020.110610
0164-1212/

et al., 2013; Yamashita and Moonen, 2013), and evolution (Abbes et al., 2011; Hall et al., 2014). They contribute negatively to software understanding and potentially lead to the introduction of flaws (Khomh et al., 2009; Khomh et al., 2012). In general, developers introduce code smells in software systems when modifications and enhancements are performed to meet new requirements. The code becomes complex and the original design is broken, lowering software quality.

Refactoring can remove code smells (Fowler et al., 1999). Refactoring is a process of improving software systems by applying transformations that should preserve their observable behavior (Wake, 2003; Kerievsky, 2004; Fowler and Beck, 2019). One of the major challenges of software-engineering research is to provide strategies for determining which refactoring to apply and when they should be applied (Fowler et al., 1999). There are many opportunities to use refactoring (Al Dallal, 2012; Bian et al., 2014; Chatzigeorgiou et al., 2017; Vedurada and Nandivada, 2017; Terra et al., 2018) to remove code smells.

However, there are many problems with refactoring and the refactoring process; among other problems: (a) How to detect smells, either about code or design? (b) After detecting these smells, which refactoring should be applied? (c) What are the steps to apply these refactoring? (d) What are the gains when applying these refactoring to remove code smells? According to Mealy and Strooper (2006) and Mealy et al. (2007), these questions, without automated support, are difficult to answer (Moha et al., 2010). Looking at the entire process and these problems, we claim that the relationship between code smells and refactoring should be further investigated.

Tertiary systematic literature reviews (SLRs) are reviews of reviews, using secondary studies, in a given area, to provide an overview of the state of the evidence in that area. The software-engineering community accepted secondary and tertiary studies as useful and helpful (Kitchenham et al., 2010b; Petersen et al., 2008). We chose to perform a tertiary study to investigate the relationship between code smells and refactoring because it is one of the ways used by researchers to provide evidence in a specific area, which can be integrated with practical experience regarding software development and maintenance. There is a relatively high number of secondary and tertiary studies in software engineering (Nurdiani et al., 2016; Garousi and Mäntylä, 2016; Hoda et al., 2017; Rios et al., 2018). Other studies (Kitchenham et al., 2010a) reported the usefulness and value of these studies.

Thus, we perform a tertiary SLR (Kitchenham et al., 2010b), from surveys, systematic mappings, and secondary SLRs, to understand and report on the relationship (or lack thereof) between code smells and refactoring. There are large numbers of studies on code smells and refactoring, respectively, evaluate their implications and contexts, usually in the form of secondary studies, to explore these topics together. In addition to the relationship between code smells and refactoring, we also study and discuss *what we know* and *what we do not know* about code smells and refactoring, such as the detection of smells (types, techniques, tools) and applications of refactoring (opportunities, tools).

Our study answers five research questions (RQs):

- *RQ1: What refactoring-related topics have been investigated in secondary studies?*
- *RQ2: What smells-related topics have been investigated in secondary studies?*
- *RQ3: Which tools have been mentioned for code smell detection and refactoring support?*
- *RQ4: Which RQs have been studied on code smells and refactoring? What are the highest cited secondary studies?*
- *RQ5: What are the annual trends of types, quality, and the number of primary studies reviewed by the secondary studies?*

We identify 40 secondary studies on code smells and refactoring. We analyze the secondary studies to identify the most discussed topics. We then explore these topics in detail, summarizing the observations and challenges related to these topics. We name an observation (*what we know*) a consensus in the literature, whereas a challenge (*what we do not know*) is a topic still open and to be better explored. Fig. 1 summarizes our research method.

Thus, we answer our research questions and provide the following contributions:

- We cross-reference the most frequent code smells with their detection approaches, detection tools, suggested refactoring, and refactoring tools (Table 8).
- We report the relationships of the top 10 code smells with their refactoring and their impact on quality (Fig. 23). Also, we relate internal attributes with external quality attributes using the QMOOD model (Bansiya and Davis, 2002). Thus, we show that refactoring affect quality more than code smells (Fig. 24).
- We present the implications of this study from the perspective of practitioners, researchers, and instructors (Section 6).
- We report on 13 open issues about code smells and refactoring (Section 7).

This paper is organized as follows: Section 2 defines code smells and refactoring. Section 3 presents the structure of our tertiary systematic literature review, with goal and RQs, identification of relevant literature, selection criteria, quality assessment, data extraction, and execution. Section 4 shows the main findings and discussion about the observations and challenges. Section 5 discusses the relationship between quality, code smells, and refactoring. Section 6 presents the implications of our research from the perspective of practitioners, researchers, and instructors. Section 7 presents open issues useful for the future works. Section 8 discusses the main threats to validity of our study. Finally, Section 9 concludes with future work.

2. Background

In this section, we present some key concepts needed to deepen the discussion about smells and refactoring.

2.1. Smells

Although “smell” is a well-known practical concept, there is not a rigorous definition nor an agreement on how to categorize it and organize it. Next, we summarize the main definitions, taxonomies, and categories related to smells.

2.1.1. Definitions

The term “smell” refers to some internal problem in the software either at a lower level, known as code level (Fowler et al., 1999) or higher, *design* (Brown et al., 1998) describing symptoms observed in components that impair software evolution. According to such level, a smell is respectively named *code smell* or *design smell*.

Differently from a bug, a smell does not necessarily cause a fault in the application but may lead to other negative consequences, impacting on software maintenance and evolution.

It is undeniable that the concept of smells was adopted, first, by the agile software development community as a way of pointing out something wrong or an improvement point (Highsmith and Fowler, 2001; Beck and Andres, 2004; Elssamadisy, 2007). Currently, the industry has also adopted this term to represent anomalies in software elements.

The use of the term “smell” became popular mainly due to the original work of Fowler et al. (1999), who used it to identify code patterns that contain structural problems and, therefore, should be

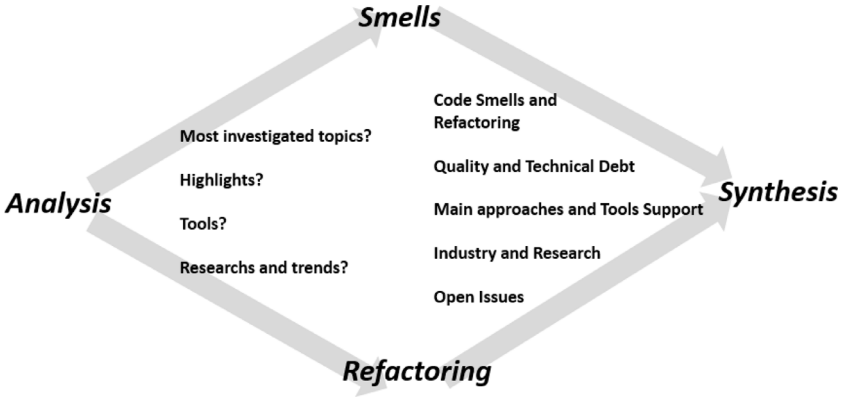

Fig. 1. The strategy used in this research, from the analysis process to the consolidation of results.

Table 1
List of design smells presented by Brown et al. (1998).

Smell	Description
Blob	As know as *God Class*, is a style of procedural design procedural which brings an object to have too many responsibilities (*Controller*) and attributes with low cohesion, while others only save data or execute simple processes
Lava Flow	Dead code and forgot information frozen with design
Functional Decomposition	A procedural code in a technology that implements the OO paradigm (usually the main function that calls many others), caused by the previous expertise of the developers in a procedural language and little experience in OO
Poltergeist	Classes that have a role and life cycle very limited, frequently starting a process for other objects
Spaghetti Code	Use of classes without structures, long methods without parameters, use of global variables, in addition to not exploiting and preventing the application of OO principles such as inheritance and polymorphism
Cut and Paste Programming	Reused code by a copy of code fragments, generating maintenance problems
Swiss Army Knife	Exposes the high complexity to meet the predictable needs of a part of the system (usually utility classes with many responsibilities)

improved. Fowler et al. (1999) were pioneers in identifying and discussing code smells and providing a practical guide to techniques to resolve them.

Brown et al. (1998) present 40 anti-patterns, which describes common occurrences for a problem that generates negative consequences. Anti-patterns are categorized in development, architecture, and project management. In Table 1, the main smells related to code are presented.

Fowler et al. (1999) described 22 smells and associated sequences of refactoring that could be applied to mitigate each smell. Such work made an important contribution by organizing and cataloging a list of smells, presented in Table 2. Recently, Fowler updated information on smells, introducing the other 6 smells to the original list (Fowler and Beck, 2019).

Wake (2003) extended that list (Table 3), taking into account some problematic aspects, often identified by software developers in practice. Kerievsky (2004) brought this discussion from the perspective of the application of design patterns. Gamma et al. proposed design patterns (Gamma et al., 1994) that provide targets for refactoring. Design Patterns represent solutions commonly used by developers to solve recurrent problems in a specific context. Kerievsky broadened that list, suggesting specific refactoring for the implementation of design patterns, and including a few more smells (Table 4).

Martin (2008) seeks to identify possible problems in code from the standpoint of cleaning heuristics, that is, what is needed in terms of style rules, good practices, and discipline to keep code clean. The smells described are not only related to the code structure, but also comments, building environment, error handling, formatting, element naming, and even testing.

Although there is an agreement concerning many smells, we can also find distinct points of view, depending on the practical experience of each author. For example, Mihancea (2006), inheritance is considered both a good practice of OO design and, at the same time, a problem for software maintenance and evolution. In other's works (Offutt et al., 2001; Mihancea and Marinescu, 2009), particular modes of use of inheritance and polymorphism it is related to comprehension pitfalls and repetitive patterns, which can easily deceive developers during the activities of software understanding and evolution.

The way usually adopted to describe smells is the original description proposed by Fowler et al. (1999). However, Zhang et al. (2008) has attempted to define a distinct approach to represent some specific smells (*e.g., Data Clumps, Middle Man, Message Chains, Speculative Generality, Switch Statements*).

Among the smells previously described, it is necessary a more detailed explanation regarding *Code Clones*. Although clone studies have grown in recent years, with specific communities dedicated to the subject, we have chosen to keep clone studies within our research. A clone (Koschke, 2007) is something that appears to be a copy of an original form. It is a synonym of duplicate. Although cloning leads to redundant code, not every redundant code is a clone. There may be cases in which two code segments that are no copy of each other happens to be similar or even identical by accident. Also, there may be redundant code that is semantically equivalent but has an entirely different implementation. There are four types of clones, describe as follows (Sheneamer and Kalita, 2016): a) *Type-1 (Exact Clone)* are the clones which look like an original code; b) *Type-2 (Renamed/parameterized Clone)* is the clones where variations come in the name of literals, keywords, variables, among others; c) *Type-3 (Near-miss Clone)* are changed to persist in the code in the form of addition, deletion, and modification of statements; and finally d) *Type-4 (Semantic Clone)* are function or behavior of the clone remains same, but the syntax or coding of the program is different.

2.1.2. Categorization

An interesting way to understand smells is through categorization, based on possible relationships between them, aiming to achieve better comprehension (Mantyla, 2003). For example,

Table 2
List of code smells presented by Fowler et al. (1999) and Fowler and Beck (2019).

Smell	Description
Duplicated Code	Consists of equal or very similar passages in different fragments of the same code base
Long Method/Long Function	Very large method/function and, therefore, difficult to understand, extend and modify. It is very likely that this method has too many responsibilities, hurting one of the principles of a good OO design (*SRP: Single Responsibility Principle* (Martin and Martin, 2007))
Large Class	Class that has many responsibilities and therefore contains many variables and methods. The same *SRP* also applies in this case
Long Parameter List	Extensive parameter list, which makes it difficult to understand and is usually an indication that the method has too many responsibilities. This smell has a strong relationship with *Long Method*
Divergent Change	A single class needs to be changed for many reasons. This is a clear indication that it is not sufficiently cohesive and must be divided
Shotgun Surgery	Opposite to *Divergent Change*, because when it happens a modification, several different classes have to be changed
Feature Envy	When a method is more interested in members of other classes than its own, is a clear sign that it is in the wrong class
Data Clumps	Data structures that always appear together, and when one of the items is not present, the whole set loses its meaning
Primitive Obsession	It represents the situation where primitive types are used in place of light classes
Switch Statements/Repeated Switches	It is not necessarily smells by definition, but when they are widely used, they are usually a sign of problems, especially when used to identify the behavior of an object based on its type
Parallel Inheritance Hierarchies	Existence of two hierarchies of classes that are fully connected, that is, when adding a subclass in one of the hierarchies, it is required that a similar subclass be created in the other
Lazy Class	Classes that do not have sufficient responsibilities and therefore should not exist
Speculative Generality	Code snippets are designed to support future software behavior that is not yet required
Temporary Field	Nember-only used in specific situations, and that outside of it has no meaning
Message Chains	One object accesses another, to then access another object belonging to this second, and so on, causing a high coupling between classes
Middle Man	Identified how much a class has almost no logic, as it delegates almost everything to another class
Inappropriate Intimacy	A case where two classes are known too, characterizing a high level of coupling
Alternative Classes with Different Interfaces	One class supports different classes, but their interface is different
Incomplete Class Library	The software uses a library that is not complete, and therefore extensions to that library are required
Data Class	The class that serves only as a container of data, without any behavior. Generally, other classes are responsible for manipulating their data, which is a case of *Feature Envy*
Refused Bequest	It indicates that a subclass does not use inherited data or behaviors
Comments	It cannot be considered a smell by definition but should be used with care as they are generally not required. Whenever it is necessary to insert a comment, it is worth checking if the code cannot be more expressive
Mysterious Name	Non-significant names that do not represent the software elements
Global Data	It can be modified from anywhere in the code base, and there's no mechanism to discover which bit of code touched it
Mutable Data	It changes to data can often lead to unexpected consequences and tricky bugs
Lazy Element	Software elements designed to grow, but do not conform with software evolution
Insider Trading	Coupling problems caused by trade data between modules

Table 3
List of code smells presented by Wake (2003).

Smell	Description
Type Embedded in Name	Names used, usually defined with duplication, such as *schedule.addCourse(course)* instead of *schedule.add(course)*. This category also included the use of Hungarian notation and variables that reflect their type in counterpoint to their purpose or function
Uncommunicative Names	Names used in software elements (usually attributes and local variables) that do not communicate their name/intent enough, such as *x* or *value1*. It is even more critical when used in methods and classes
Inconsistent Names	Same name used in different places, for different purposes
Dead Code	Characterized by a variable, attribute, or code fragment that is not used anywhere. It is usually a result of a code change with improper cleaning
Null Check	Occurrences that repeatedly appear, verifying the null values of objects
Complicated Boolean Expression	Code snippets involving boolean operators such as *and, or* and *not*
Special Case	Complex conditional statements
Magic Numbers	Numeric values that appear deliberately in the code and that invariably do not change

Wake (2003) proposed a classification of the smells cataloged by Fowler et al. (1999), with the following division:

- *Smells within Classes*: smells identified with simple metrics (*comments, long method, large class, long parameter list*), names that need to be improved (*type embedded in name, uncommunicative name, inconsistent names*), unnecessary complexity (*dead code, speculative generality*), code snippets that need to be removed (*magic numbers, duplicated code, alternative classes with different interfaces*), and problems in conditional logic (*null check, complicated boolean expression, special case, simulated inheritance*); and
- *Smells between Classes*: in this category, we find smells that represent data like lost objects, with the absence of appropriate behavior (*primitive obsession, data class, data clump, temporary field*), relationship between class hierarchies (*refused bequest, inappropriate intimacy, lazy class, combinatorial explosion*), balancing responsibilities (*feature envy, message chains, middle man*), code changes (*divergent change, shotgun surgery, parallel inheritance hierarchies*), and the lack of an incomplete library class.

Mantyla et al. (2003) proposed another taxonomy of smells, presented as follows:

- *Bloaters:* a bloater represents any element in the code that has become very large and can not be effectively handled. In general, bloaters are difficult to understand and modify. Smells belonging to this category are *Long Method, Large Class, Primitive Obsession, Long Parameter List* and *Data Clumps*;
- *Object-Orientation Abusers:* workaround solutions used in the code, without exploring principles of a good OO design (Martin and Martin, 2007). Smells in this category is *Switch*

Table 4
List of code smells presented by Kerievsky (2004).

Smell	Description
Conditional Complexity	It describes that although conditional structures are not problems in themselves, the exaggerated use of them is a smell that must be tackled
Indecent Exposure	It happens when clients have too much access to the classes they use. It unnecessarily increases the complexity of the system
Solution Sprawl	Similar to the *Shotgun Surgery*, where an update causes changes in several parts of the system
Combinatorial Explosion	It is a more subtle form, but very similar to *Duplicated Code*, where several code snippets execute the same function but in objects of different types
Oddball Solution	It occurs when there are two ways to solve the same problem on the same system, which is usually a subtle sign of *Duplicated Code*

Statements, Temporary Field, Refused Bequest, Alternative Classes with Different Interfaces and *Parallel Inheritance Hierarchies*;
- *Change Preventers:* software structures very difficult to modify; in general, this difficulty may occur at one or several points. In this category we find *Divergent Change* and *Shotgun Surgery*;
- *Dispensables:* smells that are unnecessary and, therefore, should be deleted. Smells in this category are *Duplicated Code, Lazy Class, Data Class*, and *Speculative Generality*;
- *Couplers:* smells characterizing a high coupling, like *Feature Envy* and *Inappropriate Intimacy*.

In addition to the proposed taxonomy, Mäntylä presents in another work (Mantyla, 2003) a set of metrics supporting the identification of smells, as well as a study of how effective is the use of these metrics, suggesting techniques to carry out measurements. Developers' opinions on these smells and their perceptions can vary significantly due to some factors, like experience, theoretical knowledge, and familiarity with the code in question, among others.

Perez (2011) proposes another smell classification, according to problem levels. The smells are categorized as low-level and high-level smells. The low-level smells are related to particular problems in the code, such as *Large Class* and *Long Method*. The high-level smells relate to more complex problems that may be detected in the code structure, such as *Blob* (see Table 1), for instance.

Some of the low-level smells could be considered equivalent to code smells, whereas some of the high-level smells could be regarded as equivalent to the architectural/design smells. Sometimes, high-level smells manifest by the composition of low-level smells.

In addition to several definitions of the term "smell" definitions, there are several ways to detect smells, ranging from human perception to metrics, rule-based strategies, search-based methods, and software visualization. In practice, tools are also built to support such detection mechanisms, an issue explored by some studies of our research.

2.2. Refactoring

Refactoring is the primary approach to remove smells. Next, we summarize the main concepts, the refactoring process, and some automation aspects regarding refactoring.

2.2.1. Definition

The term "refactoring" came from the work of Opdyke (1992), which defines it as reorganization strategies that support a change in a software element. Refactoring helps to make the code more readable and eliminating possible problems, as well as improving the internal quality attributes of the software (Mens and Tourwe, 2004). Refactoring is also used for reengineering, allowing to turn more modular and structured a specific system (legacy or decayed code) (Fanta and Rajlich, 1998).

There are different levels of abstraction and types of software artifacts that one can apply the refactoring. For instance, it is possible to apply refactoring in UML models, database schemas, requirements, software architecture, and structures of a language (Mens and Tourwe, 2004). So refactoring focuses not only on the source code but also on other artifacts, and for this reason, there is a need to keep all the artifacts synchronized. Because refactoring does not change the behavior of a program, the order of application of the techniques may vary according to distinct criteria. Often, some techniques can be used in sequence, as long as they satisfy the preconditions for the techniques used *a posteriori*. Other times, the sequence is arbitrary.

The refactoring typically occurs at two levels: high and low level. High-level refactoring (or composite refactoring) can be defined such as those referring to significant (usually macro or architectural) design changes, and low-level (or primitive refactoring) are those for small (ordinary) changes. The work of Opdyke (1992) also introduces a fundamental element for refactoring: the precondition, a to guarantee that the transformation preserves the program behavior. Preconditions are checked before applying the transformation to make sure that it will not introduce compilation problems or change the behavior of the program. For example, the *Extract Method refactoring* checks the selected code fragment contains broken elements before to perform the refactoring. Opdyke states, to perform high-level refactoring, it is necessary to perform low-level refactoring.

The point is that low-level refactoring are rarely executed in isolation. Generally, they are used together, when the developers already have in mind a defined objective to apply the techniques to achieve the desired design. The pioneer thesis of Opdyke (1992) defined 23 primitive refactoring techniques and presented three examples of composite refactoring, formed by primitive techniques. Since then, a lot of work (López et al., 2014; Lahtinen and Leppänen, 2016; Haendler and Frysak, 2018) has been made to improve refactoring adoption, mainly concerning the refactoring process and automation.

2.2.2. Process and automation

Every refactoring can be composed of a set of simple basic steps. If a developer does not know where to start or if she feels overwhelmed, these basic steps are a good way to start. Sometimes such a set of basic steps is considered as a process called *refactoring process*. Mens and Tourwe (2004) identified a common process in which refactoring operations:

1. Detect pieces of code with refactoring opportunities;
2. Determine which refactoring can apply in the selected code snippet;
3. Ensure that the selected refactoring preserve the behavior;
4. Apply the chosen refactoring in the respective locations;
5. Assess the effect of the refactoring on quality characteristics of the software;
6. Maintain the consistency between the refactored artifact and other software artifacts.

At the same time, refactoring can become a continuous improvement tool for software, especially if the team is made up of developers concerned about the quality of the implemented code.

According to Parnin et al. (2008), one problem-related to refactoring is the benefits of software quality gained through these practices are often diluted by the high costs and low priority when compared to the urgency of bug fixes and the implementation of new features.

This problem is because 40% of the time invested in software maintenance is the cost to understand the software and architecture will evolve (Telea and Voinea, 2011).

Murphy-Hill and Black (2008) presents two terms that can summarize the posture concerning refactoring: *floss refactoring*, that is, to adopt day-to-day refactoring techniques in a healthy and disciplined way and *root channel refactoring* when there is no habit of cleaning and this can be very costly over time, with the necessity to plan.

Although the refactoring scenario initially was proposed for object-oriented languages, other researchers have applied the idea of refactoring to various paradigms, such as functional (Li et al., 2008), logic programming (Saadeh and Kourie, 2009), aspects (Piveta et al., 2006), among others. The general idea is similar to that of object-oriented languages, but the conditions and mechanics are different.

Roberts (1999), another pioneer in this field, extended the work of Opdyke, addressing the optics of creating refactoring support tools that are fast and reliable for developers. For this, Roberts added the post-conditional element in refactoring, which describes what should or should not be valid after the application of the technique. The *Refactoring Browser* tool, created by Roberts, implements some refactorings inside the Smalltalk programming language.

Some tools proposed for various programming languages, such as Java, Smalltalk, C++, C#, Python, and others. The goal is double: (i) to reduce possible errors and code inconsistencies and (ii) to automate the refactoring's practice. Through the tool, the developer can select the code snippet, which technique should be applied, and which parameters required for execution. The tool automatically checks the preconditions and, if all is correct, uses refactoring. Some tools, whether commercial or academic, also have been proposed to support refactoring in the form of IDE plugins.

For the Java language, there is Tsantalis (2018) and JRefactory (2018). Refactory (2013) allows developers to apply refactorings from UML diagrams. CppRefactory (2001) is an open-source refactoring tool that automates the refactoring process in C++ projects. For C#, Software (2018), Express (2018), and JetBrains (2018). XRefactory (1998) is a refactoring *browser* for Emacs, XEmacs, and jEdit. For Visual Basic, the tool is Aivosto (2018). And, for Python, the tools are Rope (2018) and the Man (2018).

Tsantalis (2010) advocates the automatic detection approach. Thus, his research group analyzes *JDeodorant* as a tool that automatically detects smells. Tsantalis and Chatzigeorgiou (2011) proposed to analyze the repository of code versions, to classify the refactoring according to the number, proximity, and extent of the changes crossing with the corresponding smells. Another approach based on the selection of techniques using technical debt as a metric in which the debt and interest rate to pay are calculated (Zazworka et al., 2011). Of course, it is necessary to have some mechanism to select and rank the refactoring techniques (Piveta, 2009; Bruch et al., 2009; Lee et al., 2013).

3. Study design

We realized a tertiary systematic literature review (SLR), adopting the Kitchenham et al. (2007, 2009, 2010b) SLR guidelines. The SLR followed five main steps (Fig. 2): (1) definition of goal and research questions; (2) identification of relevant papers; (3) selection criteria, (4) quality assessment, and (5) data extraction. These steps are detailed as follows.

3.1. Goal and research questions

The main goal of this work is to examine the current research works and the most important contributions of the smells and refactoring fields. At the same time, a comprehensive and systematic view can be a contribution to the research community as it can facilitate assessments and discussions of future directions related to these issues.

To structure our research, we studied and evaluated other tertiary studies (Imtiaz et al., 2013; Hoda et al., 2017; Khan et al., 2019). We also wanted to understand how the research area evolved. Thus, we studied the research questions (RQs) asked in other tertiary studies in SE (Kitchenham et al., 2010b; Garousi and Mäntylä, 2016; Rios et al., 2018) to inspire our work. We built the RQs using this information.

The purpose of this SLR is to answer the following RQs:

- *RQ1: What refactoring-related topics have been investigated in secondary studies?*
- *RQ2: What smells-related topics have been investigated in secondary studies?* Answering RQ1 and RQ2 will enable us to determine the refactoring and smells topics covered and not covered by secondary studies. Knowing the topics not covered will pinpoint the need for conducting secondary studies in those topics.
- *RQ3: Which tools have been mentioned for code smell detection and refactoring support?* Answering RQ3, we are investigating which tools have been mentioned to aim to smell detection. Furthermore, we are also searching for supporting tools for refactoring.
- *RQ4: Which RQs have been studied on code smells and refactoring? What are the highest cited secondary studies?* Responding to RQ4, we analyze which aspects discussed in the studies and what the correlation between them is. Given the importance of citations to determine scientific merit, we decided to investigate what secondary studies are the most cited.
- *RQ5: What are the annual trends of types, quality, and the number of primary studies reviewed by the secondary studies?* Answering RQ5, will allow us to get a big picture of the landscape in this field and on the several studies about it.

3.2. Identification of relevant literature

The search involved in eight digital libraries (see Table 5), aiming to identify relevant secondary studies, published in journals, and conferences about software engineering, software development, software maintenance and evolution, and software quality.

We used the PICOC process, the search string construction, the search engines, and the selection criteria for the returned studies. Each of them is described hereafter. PICOC is a process that aids in structuring SLRs (Kitchenham et al., 2007; 2009; 2010b). This process consists of identifying the population (P), the intervention (I), the comparison (C), the expected outcomes (O), and the (C) context of a SLR. They are:

- Population (P): Conferences and journals about Software engineering, software development, software maintenance and evolution, software quality;
- Intervention (I): Systematic literature, systematic mapping, survey, systematic literature review;
- Comparison (C): Not applicable, since the purpose of the study is to characterize the secondary studies available in the literature;

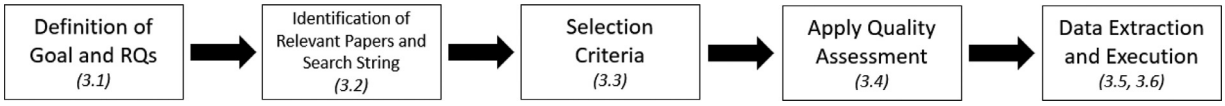

Fig. 2. Steps defined of SLR.

Table 5
Digital libraries, criteria considered, and search queries.

Database	URL	Criteria	Query
ACM Digital Library	http://dl.acm.org/	Published since 1993, Content Format: PDF	acmdlTitle:(smell refactoring) AND recordAbstract:(survey ''systematic literature'' ''systematic literature review'' ''systematic review'' review ''systematic mapping'' ''systematic study'' ''mapping study'') AND (tool* technique* method* practice* problem*)
IEEE Xplore	http://ieeexplore.ieee.org/Xplore/home.jsp	Filters Applied: 1992–2018, Conferences Journals and Magazines	(((''Document Title'':smell OR refactoring) AND (''Abstract'':systematic literature OR systematic mapping OR mapping study OR systematic review OR survey) AND (''Publication Title'':software engineering OR software quality OR software maintenance OR software evolution))
Science Direct	http://www.sciencedirect.com/	Year 1992–2018, Find articles with these terms (1), Title, abstract and keywords (2), Publication title: Journal of Systems and Software; Information and Software Technology	(1) smell OR refactoring (2) survey OR ''systematic literature'' OR ''systematic literature review'' OR ''systematic review'' OR review OR ''systematic mapping'' OR ''systematic study'' OR ''mapping study''
Wiley InterScience	http://www.interscience.wiley.com	Date Range: 01/1992 and 12/2018, Computer Science	survey OR 'systematic literature' OR 'systematic review' OR review OR 'systematic mapping' OR 'systematic study' OR study OR mapping'' in Abstract and ''refactoring OR smell'' in Abstract AND software AND (technique* OR tool* OR method* OR practice* OR application OR problem)
Scopus	https://www.scopus.com/	English, Computer Science Software Engineering, Conference Paper and Chapter, before to 1992	(TITLE(survey OR ''systematic literature'' OR ''systematic literature review'' OR ''systematic review'' OR review OR ''systematic mapping'' OR ''systematic study'' OR ''mapping study'') AND TITLE-ABS(refactoring OR smell)) AND ALL(tool* OR technique* OR method* OR practice* OR problem* AND software) AND ((PUBYEAR > 1992) AND (PUBYEAR < 2019)) AND (LIMIT-TO(SUBJAREA, ''COMP'')) AND (LIMIT-TO(DOCTYPE, ''cp'') OR LIMIT-TO(DOCTYPE, ''ar'')) AND (LIMIT-TO(LANGUAGE, ''English''))
AIS eLibrary	http://aisel.aisnet.org/	Date Range: 1992-01-01 and 2019-01-01, Limited search to: All Repositories, Format: Links, Computer Sciences	(survey OR 'systematic literature' OR 'systematic review' OR review OR 'systematic mapping' AND software) AND abstract:(refactoring OR smell) AND (software AND (technique* OR tool* OR method* OR practice* OR application OR problem*))
Google Scholar	https://scholar.google.com	1992–2018	allintitle:(smell OR refactoring) AND (''systematic literature'' OR ''systematic mapping'' OR ''systematic study'' OR ''literature review'' OR survey)
Springer	http://link.springer.com/	English, Computer Science, Software Engineering, Conference Paper and Chapter, 1992 and 2018	(systematic literature OR mapping study OR systematic mapping OR literature review) AND (smell OR refactor*) AND (tool* OR technique* OR method* OR practice* OR problem*)

- Outcomes (O): Methods, techniques, practices, application, problems, strategies, and tools described in Surveys, Systematic Mappings, and Systematic Literature Reviews;
- Context (C): Domain of smells and refactoring.

The strategies for the final search string were: (a) derivation of terms used in the research question (example: *smells, refactoring*) and related to the RQs; (b) list of keywords of papers consulted; (c) use of the Boolean operator OR to incorporate synonyms; (d) use of the AND operator to make the conjunction between the differ-

ent keywords. The search string was built based on the following terms:

survey, systematic mapping, systematic literature review, smell, refactoring, tool, technique, method, practice, application, problem, software

The digital libraries chosen are presented in Table 5. We build specific search strings to each digital library, taking into account its characteristics. Some criteria have been configured in the search engine itself. The terms used in the queries were also prioritized, depending on each search engine in the databases.

3.3. Selection criteria

To select the studies returned with the search strings, we elaborate on a list of criteria for exclusion and inclusion.

The exclusion criteria used were as follows:

- Articles not related to the software engineering area (development, quality, maintenance, and evolution of software);
- Articles/studies not written in English;
- Works presented in non-academic events in the area of computing (*e.g., Agile Conference, Agiles Conf*);
- studies such as tutorials, position papers, theses, and dissertations.

The criteria for inclusion include:

- Secondary studies (Systematic Literature Reviews, Systematic Mappings, and Surveys) about smells and refactoring;
- Articles describing the use/development/evaluation of tools, methods, practices for smells detection;
- Articles describing the use/development/evaluation of refactoring tools, methods and techniques;
- Works published between 1992 and 2018. We defined 1992 as the beginning of the research, because of the doctoral thesis published by Opdyke (1992), the first detailed written work on refactoring (Fowler and Beck, 2019);
- Only full papers (more than 6 pages);
- Be available online for download.

To avoid missing any potentially relevant studies then we applied the snowballing technique by checking the references of each selected study (Wohlin, 2014).

3.4. Quality assessment

Each candidate Survey, Systematic Mapping, or Systematic Literature Review was evaluated using the same criteria adopted by previous research studies (*e.g.,* by Kitchenham et al.) in tertiary studies (Kitchenham et al., 2009; 2010b). These criteria were defined by the Centre for Reviews and Dissemination (CDR) Database of Abstracts of Reviews of Effects (DARE), of the York University (University of York, 2018). The criteria are four quality assessment questions, described as follows:

1. Are the reviews inclusion and exclusion criteria described and appropriate?
2. Is the literature search likely to have covered all relevant studies?
3. Did the reviewers assess the quality/validity of the included studies?
4. Were the primary data/studies adequately described?

Kitchenham et al. (2009, 2010b) proposed a score for these questions. For each candidate secondary study in our pool, the quality score was calculated by assigning {0, 0.5, 1} to each of the four questions and then adding them up.

3.5. Data extraction

We structured the Google Forms for data extraction. In this form, we find the main information that we consider relevant regarding the papers. In general, we consider:

- Paper's information: title, authors, authorś institution and country, year of publication, initial and final year of research, where the paper published, abstract and keywords, type of publication (journal or conference);
- Main contributions, evidence, and type of method research (Survey, Systematic Mapping, or Systematic Literature Review);
- Databases used and Research Questions defined;
- Amount of papers considered and analyzed;
- Categorization of smells research (definition, detection options, support tools, technical debt, and others) and categorization of refactoring research (techniques, opportunities, support tools, tests, and others);
- Research approaches (case studies, surveys, experimental and empirical studies);
- Type of projects (FLOSS, commercial, toy/academic) and repositories and programming languages used;
- Context: refactoring techniques presented, tools more cited, and smells (design/code) described.

In addition to the spreadsheet, Mendeley[1] is also used to assist in the cataloging, structuring and searching for papers. All artifacts produced from our research are available on the replication package (Lacerda, 2019).

3.6. Execution

With the research protocol defined, we started filtering these studies. As there were a large number of papers identified in the search phase, the filtering process consisted of four steps. Each step used the inclusion and exclusion criteria, and relevance of the study according to its content. We describe these steps as follows:

1. Search and delete studies based on criteria defined through reading the title, abstract and keywords;
2. Remove duplicate papers and full-text analysis using inclusion/exclusion criteria;
3. For selected studies, we apply the snowballing process;
4. Define the context and categorization of the works, saving this information using the adopted tools (spreadsheet and Mendeley).

In total, there were 467 secondary studies, and we selected 59 studies. Of these 59, 26 (six duplicates and 20 based on the selection criteria) were discarded, leaving 33 selected secondary studies. After applying to snowball, another seven studies were added, totaling 40 selected studies. For each paper selected, the data were extracted and analyzed (Fig. 3).

Considering all databases, we obtained an average accuracy of 12.63% in the search, in which Google Scholar showed better accuracy (33.33%), while Springer had the lowest accuracy (5.62%).

In the selected secondary studies, 19 different databases (Fig. 4) were used for research, highlighting IEEE Xplore, used in 90% of the selected studies, followed by ACM Digital Library and Science Direct (80% and 75% respectively).

Selecting all these libraries together can lead to overlapping results, which requires the identification and removal of redundant results; however, this selection of libraries increases confidence in the completeness of the review. Our search considered all years between 1992 and 2018 to increase the comprehensiveness of the

[1] http://www.mendeley.com/.

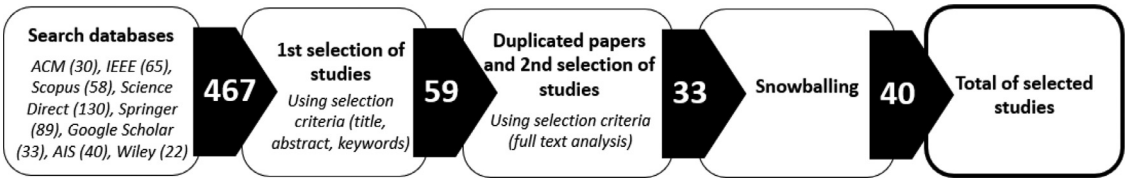

Fig. 3. Steps of execution research and number of selected studies on each databases.

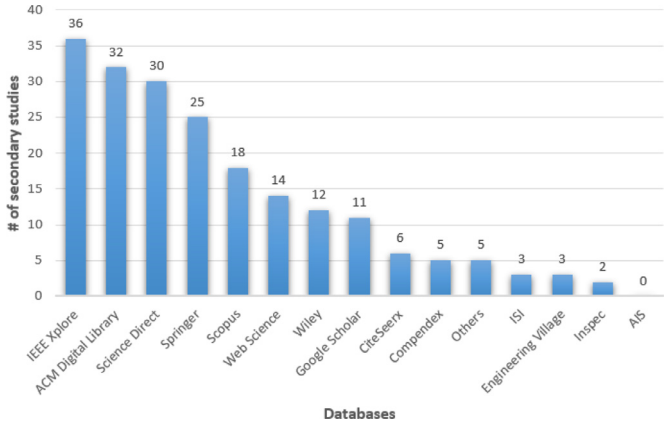

Fig. 4. Databases used in the secondary studies.

review, considering 1992 as a mark of the year of pioneer publication about refactoring (Opdyke, 1992). The list of selected secondary studies found is presented in Table 6.

We queried smells and refactoring separately to find secondary studies from these two fields because, although they are closely related, researches have studied smell and refactoring separately. For better distribution, we organize these studies into themes like refactoring (13 studies), smells (25 studies), and both (2 studies covering both themes).

Within each theme, papers are categorized into topics. As we show in Fig. 1, the topics emerged from the analyzed studies. To organize the topics, we used a card sorting technique (Barrett and Edwards, 1995; Sheikh et al., 2011). Card sorting is a knowledge elicitation method commonly used for capturing information about different ways of representing domain knowledge. It has been used in various fields such as psychology, knowledge engineering, and software engineering. We discuss the most cited topics in detail in Section 4[2]. However, We will not discuss the topics were having already distinct research communities, such as refactoring models and clones. Also, some topics that not appeared in secondary studies maybe nowadays considered as essential by the academical community: thus, a preliminary discussion of some implications of our work to the academic community is presented in Section 8.

The topics related to refactoring are trends and challenges, quality and object-orientation (OO), process, software product lines (SPL), search-based and technical debt (TD), and models. The topics related to smells are relationship with design patterns, detection tools, SPLs, clones, definitions, challenges, tests, mining, technical debt (TD), and impact and effects. The topic covered by both refactoring and smells is related to object-orientation. In Fig. 5, we present a mind-map with this distribution and quantity by topics. Some abstract topics such as models and SPL are not discussed here, because our research is more focused on code-related topics like code smells and refactoring.

In Table 7, the list of studies by publication source and venue (Journal, Conference) is displayed. The emphasis is on Systems and Software, Information and Software Technology, and Transactions on Software Engineering, which receive secondary studies in their submissions.

4. Findings

We compile the results from 40 selected studies to show which approaches used to detect code smells, which tools have supported this detection process, which refactoring techniques used and which are being supported by tools. Table 8 summarizes the ten most-cited smell, with the following information: the main approaches to smell detection, the most cited smell detection tools, the most suggested refactoring for each smell, and most cited refactoring tools. We sorted each item by the number of references. Thus, we organized the main findings to help researchers and practitioners to address their research activities (we discuss some implications about practitioners, researchers, and instructors in the Section 6). *Duplicated Code* was already the smell considered by the agile community as being the most cited smell (Fowler et al., 1999; Kerievsky, 2004; Feathers, 2004; Martin and Martin, 2007; Martin, 2008). Also, it was the most smell studied and the most referenced smell in secondary studies [S2, S4, S11, S15, S18, S24]. Here, we grouped *Code Clones* with *Duplicated Code* following the classification proposed by some works [S9, S19, S20, S30, S31, S32, S33]. Alternatively, we show in Table 9, the tools discussed by the secondary studies (studies S2 and S40 are the ones that most cited tools).

Next, we present our findings in subsections grouped by RQs.

4.1. RQ1: What refactoring-related topics have been investigated in secondary studies?

In our research, as shown in Fig. 5, we found 13 secondary studies related to refactoring [S3, S5, S6, S7, S10, S14, S17, S21, S22, S25, S27, S29, S36].

We categorize the refactoring topics of such 13 studies as they were discussed in the literature, plotting the Fig. 6. A summary of the main highlights found in these studies and a discussion of some points are presented here.

4.1.1. Refactoring techniques highlights

Ten studies mention refactoring techniques [S4, S7, S8, S10, S13, S14, S22, S25, S30, S31]. We presented the top 10 refactoring more quoted on studies (Fig. 7). The techniques that **most appear are extraction techniques (*e.g.*,method, variable, class)** [S4, S7, S8, S10, S13, S14, S22, S25, S30].

Extraction techniques are quoted as a technique used in 5 smells more cited (Table 8). For instance, *Extract Class* can be applied in *Duplicated Code/Clones, Large Class/God Class, Divergent Change*, and *Data Clumps*. We note that the same refactoring can

[2] We make all categories and topics and their organization in other documents used during the research, available at Lacerda (2019).

Table 6
List of selected secondary studies.

#	Title	Ref
S1	A systematic literature mapping on the relationship between design patterns and bad smells	Sousa et al. (2018)
S2	A review-based comparative study of bad smell detection tools	Fernandes et al. (2016)
S3	UML model refactoring: a systematic literature review	Misbhauddin and Alshayeb (2015)
S4	Identifying Various Code-Smells and Refactoring Opportunities in Object-Oriented Software System : A systematic Literature Review	Singh and Kumar (2018)
S5	Trends, Opportunities and Challenges of Software Refactoring: A Systematic Literature Review	Abebe and Yoo (2014b)
S6	Classification and Summarization of Software Refactoring Researches: A Literature Review Approach	Abebe and Yoo (2014a)
S7	Empirical Evaluation of the Impact of Object-Oriented Code Refactoring on Quality Attributes: A Systematic Literature Review	Al Dallal and Abdin (2018)
S8	Bad Smells in Software Product Lines: A Systematic Review	Vale et al. (2014)
S9	The vision of software clone management: Past, present, and future	Roy et al. (2014)
S10	A survey of software refactoring	Mens and Tourwe (2004)
S11	Code Bad Smells: a review of current knowledge	Zhang et al. (2011)
S12	A Systematic Literature Review: Code Bad Smells in Java Source Code	Gupta et al. (2017)
S13	A systematic literature review: Refactoring for disclosing code smells in object oriented software	Singh and Kaur (2017)
S14	A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring	Laguna and Crespo (2013)
S15	A survey on software smells	Sharma and Spinellis (2018)
S16	Smells in software test code: A survey of knowledge in industry and academia	Garousi and Küçük (2018)
S17	A survey of search-based refactoring for software maintenance	Mohan and Greer (2018)
S18	A review of code smell mining techniques	Rasool and Arshad (2015)
S19	Clone evolution: a systematic review	Pate et al. (2013)
S20	Software clone detection: A systematic review	Rattan et al. (2013)
S21	Managing architectural technical debt: A unified model and systematic literature review	Besker et al. (2018)
S22	Identifying refactoring opportunities in object-oriented code: A systematic literature review	Al Dallal (2015)
S23	Identification and management of technical debt: A systematic mapping study	Alves et al. (2016)
S24	A systematic review on the code smell effect	Santos et al. (2018)
S25	A systematic review on search-based refactoring	Mariani and Vergilio (2017)
S26	A systematic mapping study on technical debt and its management	Li et al. (2015)
S27	Analyzing the concept of technical debt in the context of agile software development: A systematic literature review	Behutiye et al. (2017)
S28	Co-ocurrence of Design Patterns and Bad Smells in Software Systems : A Systematic Literature Review	Cardoso and Figueiredo (2015)
S29	Non-Source Code Refactoring: A Systematic Literature Review	Rochimah et al. (2015)
S30	Survey of Research on Software Clones	Koschke (2007)
S31	A Survey of Software Clone Detection Techniques	Sheneamer and Kalita (2016)
S32	A Systematic Literature Review of Code Clone Prevention Approaches	Ali and Sulaiman (2014)
S33	Systematic Mapping Study of Metrics based Clone Detection Techniques	Rattan and Kaur (2016)
S34	A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems	Sabir et al. (2018)
S35	The Impact of Code Smells on Software Bugs: a Systematic Literature Review	Cairo et al. (2018)
S36	Refactoring UML Models of Object-Oriented Software: A Systematic Review	Sidhu et al. (2018)
S37	Impact of Code Smells on the Rate of Defects in Software: A Literature Review	GRADIŠNIK, MITJA and HERIČKO (2018)
S38	Empirical evidence of code decay: A systematic mapping study	Bandi et al. (2013)
S39	Software Design Smell Detection: a systematic mapping study	Alkharabsheh et al. (2018)
S40	A systematic literature review on bad smells - 5 W's: which, when, what, who, where	d. P. Sobrinho et al. (2018)

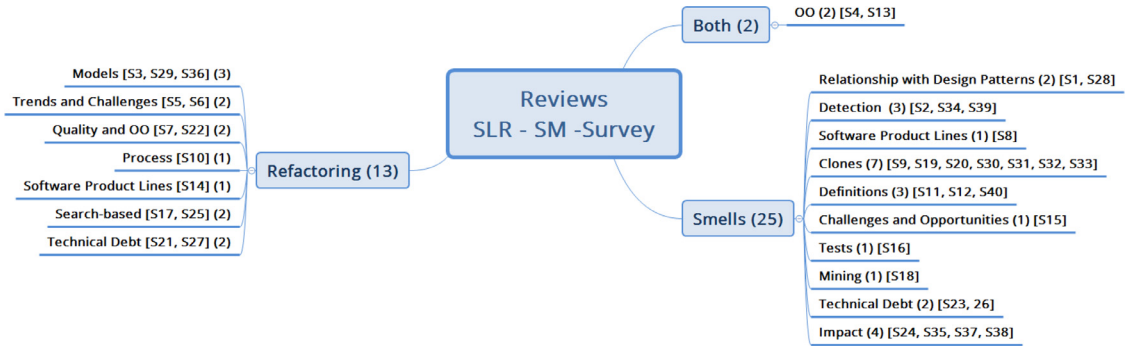

Fig. 5. Mind-map of secondary studies, categorized by smells, refactoring and both (in the parentheses are the numbers of studies and in the brackets their references).

be applied to more than one smell. Of course, in these cases, the developer should take context into account.

Also, the most cited refactoring are also the most studied, according to [S5, S7, S22, S25]. These studies show that the researchers are more interested in the *Extract Class* and *Extract Method* than in other refactoring techniques. *Move Method* also deserves a highlight. The high interest in these techniques may indicate their significant importance in the software industry. However, although these techniques could be potentially more frequently applied during the refactoring process than other refactoring activities due to their influence [S7, S22], *Extract Class* and *Extract Superclass* is rarely used in practice, as also shown in [S22]. Other techniques such as *Rename Field, Rename Method, Inline Temp*, and *Add Parameter* are among the most used techniques in practice. Still,

Table 7
Distribution of studies by venues.

Source	Venue	Amount	Studies
Systems and Software	Journal	6	S15, S16, S20, S21, S24, S26
Information and Software Technology	Journal	5	S20, S22, S23, S25, S27
IEEE Transactions on Software Engineering	Journal	3	S7, S10, S40
Journal of Software: Evolution and Process	Journal	2	S18, S19
CSMR-WCRE (IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering)	Conference	2	S9, S38
Advanced Science and Technology Letters	Journal	1	S6
Empirical Software Engineering	Journal	1	S3
International Journal on Future Revolution in Computer Science & Communication Engineering	Journal	1	S4
International Journal of Software Engineering and Its Applications	Journal	1	S5
Journal of Software Maintenance and Evolution: Research and Practice	Journal	1	S11
Journal of Software Engineering Research and Development	Journal	1	S17
Journal of Software Engineering Research and Development	Journal	1	S17
Science of Computer Programming	Journal	1	S14
International Journal of Software Engineering and Its Applications	Journal	1	S29
International Journal of Computer Applications	Journal	1	S31
International Journal of Software Engineering and Technology	Journal	1	S32
Journal of Software: Practice and Experience	Journal	1	S34
Journal of Information	Journal	1	S35
Journal of Software Quality	Journal	1	S39
IJSEKE (International Journal of Software Engineering and Knowledge Engineering)	Journal	1	S36
SAC (Symposium of Applied Computing)	Conference	1	S1
EASE (International Conference on Evaluation and Assessment in Software Engineering)	Conference	1	S2
SBCARS (Brazilian Symposium on Software Components, Architectures and Reuse)	Conference	1	S8
SBSI (Brazilian Symposium on Information Systems	Conference	1	S28
IBIF (Internationales Begegnungs und Forschungszentrum fur Informatik)	Conference	1	S30
Ain Shams Engineering Journal	Journal	1	S13
ICCSA (International Conference on Computational Science and its Applications)	Conference	1	S12
AICTC (International Conference on Advances in Information Communication Technology & Computing)	Conference	1	S33
SQAMIA (Workshop of Software Quality)	Conference	1	S37

Table 8
List of Top-10 smells and strategies to identify/remove/mitigate.

Smells	Main approaches to detect	Most cited detection tools	Most suggested refactoring	Most cited refactoring tools
Duplicated Code/Clones	**textual** [S2, S19, S20, S30, S31, S32, S33], **token** [S2, S19, S20, S30, S31, S32], **metrics-based** [S2, S18, S20, S30, S32, S33], tree [S2, S20, S31, S32, S33], strategies/rules [S2, S20, S31], probabilistic/search-based [S20, S31, S32, S33], visualization [S9, S30, S31]	**CCFinder** [S2, S9, S19, S20, S30, S31], **CloneDr** [S2,S9, S20, S31], **Nicad** [S2, S20, S31, S40], **JCD** [S2, S30, S31], PMD [S2, S9, S40], Checkstyle [S2, S18, S40], CloneDigger [S2, S9], Duploc [S20, S31], inFusion [S2, S18, S39, S40], CP-Miner [S19, S31], Jplag [S30, S31], ConQAT/CloneDectetive [S9, S20, S40], DECKARD [S19, S20], Clever [S9, S19], DECOR [S13, S39], inCode [S2, S39, S40], iPlasma [S2, S40], Gendarme [S2], Clone Miner [S9], SYMake [S2], CloneBoard [S9], CPC [S9], CloneScape[S9], SHINOBI [S9], JSync [S9], Cleman [S9], CeDAR [S9], CloneTracker [S19], Cyclone [S19], CloneInspector [S19], Columbus [S19], Datrix [S19], SimScan [S19], Simian [S19, S20], SmallDude [S19], iClones [S19], CLAN/Covet [S20, S31], ARIES [S33], JCodeCanine [S13], JDev [S13], JCCD, SAME, Borland Together, SonarQube, Stench Blossom, InsRefactor [S40]	**Extract Method** [S4, S7, S9], **Pull Up Method** [S4, S9], **Rename Method** [S4, S9], **Replace Constructor With Factory** [S4, S30], Extract Class [S4], Form Template Method [S4], Push Down Method [S9], Push Up Method [S4], Substitute Algorithm [S13], Move Method [S9], Extract Superclass [S9], Extract utility-class [S9]	**JDeodorant** [S39, S40], Wrangler [S2], CodeRush [S9]
Large Class/God Class	**metrics-based** [S2, S15, S18], **strategies/rules** [S2, S15, S38], probabilistic/search-based [S15, S38], history-based [S15], optimization-based [S15], visualization [S13]	**DECOR** [S2, S18, S38, S39, S40], PMD [S2, S18, S39, S40], Gendarme [S2], inCode [S2, S39, S40], inFusion [S2, S39, S40], iPlasma [S2, S39, S40], Checkstyle [S2, S18, S40], SDMetrics [S2, S40], Weka [S13], HIST [S4, S40], Stench Blossom [S13, S40], BSDT [S13], CodeNose [S18], JDev [S13], JCosmo [S13], 2D-DSL, NosePrints, Prodetection, P-EA, BLOP, History Miner, BBT, TACO, SMURF, EvolutionAnalyzer, Van, Borland Together, Understand, Pascal Analyzer, SCOOP/Organic, CodeVizard, IYC, MuLATo, SpIRIT, InsRefactor, JCodeOdor, JSNOSE, HULK, Paprika, Metrics, SourceMiner [S40]	**Extract Class** [S4, S13], **Extract Subclass** [S4, S13], Replace Data Value with Object [S4], Extract Interface [S4], Duplicate Observed Data [S13]	**JDeodorant** [S2, S18, S39, S40], TrueRefactor [S17]

(*continued on next page*)

Table 8 (*continued*)

Smells	Main approaches to detect	Most cited detection tools	Most suggested refactoring	Most cited refactoring tools
Feature Envy	**strategies/rules** [S2, S13, S15], **metrics-based** [S13, S18, S38], history-based [S15], optimization-based [S15], probabilistic/search-based [S13, S15], visualization [S13]	**iPlasma** [S2, S13, S39, S40], IntelliJ IDEA [S2], Stench Blossom [S13, S40], JSpIRIT [S2, S40], NosePrints [S2, S40], Weka [S13], HIST [S4, S40], JCosmo [S13], SACSEA [S13], JCodeCanine [S13], inFusion [S18, S39, S40], inCode [S13, S39, S40], CodeNose [S18], DECOR, Prodetection, P-EA, BLOP, TACO, Fluid Tool, Methodbook, Borland Together, Understand, SCOOP/Organic, MuLATo, SourceMiner [S40]	Move Method [S4, S13], **Extract Method** [S4, S7, S13], Move Field [S4]	**JDeodorant** [S13, S18, S40], JMove [S40]
Long Method	**metrics-based** [S2, S15, S18], **strategies/rules** [S2, S13, S15], probabilistic/search-based [S15]	**Checkstyle** [S2, S13, S18, S40], **PMD** [S2, S18, S39, S40], TACO [S24], Stench Blossom, **DECOR** [S18, S39, S40], inFusion, JSpIRIT, CodeNose [S18], Gendarme [S2], iPlasma [S39, S40], inCode, 2D-DSL, NosePrints, Prodetection, TACO, Teamscale, Borland Together, Understand, SCOOP/Organic, IYC, MuLATo, InsRefactor, ConQAT [S40]	**Extract Method** [S4, S7, S13], **Replace Temp with Query** [S4, S7, S13], Introduce Parameter Object and Preserve Whole Object [S13], Replace Method with Method Objects [S7, S13], Decompositional Objects [S7, S13]	**JDeodorant** [S2, S13, S18, S39, 40], TrueRefactor [S2, S17]
Long Parameter List	**strategies/rules** [S2, S13, S15], **metrics-based** [S15, S18], optimization-based [S15]	**PMD** [S2, S13, S18, S39, S40], Checkstyle [S2, S18, S40], DECOR [S18, S39, S40], CodeNose [S18], JDev [S13], SACSEA [S13], iPlasma [S39, S40], inCode, inFusion, 2D-DSL, NosePrints, P-EA, BLOP, Borland Together, Understand, Pascal Analyzer, SCOOP/Organic, MuLATo, SpIRIT, InsRefactor, JSNOSE, Metrics, SDMetrics [S40]	**Replace Parameter with Method** [S4, S13], Preserve the Whole Object [S4], Introduce Parameter Object [S4]	JDeodorant [S40]
Divergent Change	**strategies/rules** [S13, S15], metrics-based [S18], history-based [S15]	**HIST** [S4, S13, S40], DECOR [S39], Borland Together, Understand, inFusion, inCode, SCOOP/Organic, MuLATo, SourceMiner [S40]	**Extract Class** [S4, S13]	JDeodorant [S40]
Data Clumps	**metrics-based** [S2, S18], strategies/rules [S2], tree [S2], visualization [S13]	CBSDetector [S2, S40], **inCode** [S2, S39, S40], **inFusion** [S2, S39, S40], IntelliJ IDEA [S2], Stench Blossom [S2, S40], NosePrints, Borland Together [S40]	Introduce Parameter Object [S4, S13], Extract Class [S4, S13], Preserve Whole Object [S4, S13]	*
Refused Bequest	**metrics-based** [S13, S15, S18], strategies/rules [S15], visualization [S13]	**iPlasma** [S13, S39, S40], **inCode** [S2, S39, S40], **inFusion** [S2, S39, S40], IntelliJ IDEA [S2], Stench Blossom [S2], DECOR [S39, S40], 2D-DSL, NosePrints, Prodetection, Borland Together, SpIRIT [S40]	**Replace Inheritance with Delegation** [S4, S13], Push Down Method [S13], Push Down Field [S13]	*
Shotgun Surgery	**metrics-based** [S15, S18], strategies/rules [S13, S38], optimization-based [S15], history-based [S15]	**HIST** [S4, S13, S40], inFusion, inCode, iPlasma [S39, S40], DECOR [S39, S40], Prodetection, P-EA, BBT, Borland Together, Understand, SCOOP/Organic, CodeVizard, MuLATo, SpIRIT, JCodeOdor [S40]	**Move Method** [S4, S13],Move Field [S4], Inline Class [S4], Move Class [S13]	TrueRefactor [S17]
Lazy Class	metrics-based [S18]	BSDT [S13], DECOR [S39]	**Collapse Hierarchy** [S4, S13], Inline Class [S4, S13]	TrueRefactor [S2, S17]

surprisingly we did not find studies that highlight opportunities for their appropriate application. Still, *Rename Method* is the most commonly used automated refactoring [S22].

Most of the refactoring described in the studies are the same defined by Fowler et al. (1999). However, the **number of techniques explored is still small**. Indeed, the studies [S7, S25] report a low number of considered techniques (20 and 27, respectively).

In practice, it is challenging for the developer to identify refactoring opportunities, that is, to determine which type of refactoring should be applied to correct a smell [S25]. The **relationship between smells and refactoring is not a one to one relationship** [S7, S25]. We can apply more than one refactoring technique to a smell. It may even be necessary to combine more than one refactoring to remove it or reduce its impact. Also, some refactoring can be applied in more than one smell.

These observations suggest that **there is a gap between refactoring practice and research for the topic of identifying refactoring opportunities**. These results point up opportunities to evaluate unexplored or underutilized refactoring, which could (or not) be applied together.

4.1.2. Refactoring opportunities

The **refactoring opportunities, application of refactoring and tools support are the most studied** topics [S3, S5, S7, S10, S14, S17, S21, S22, S25, S27].

The most commonly used approaches to identifying refactoring opportunities are quality metrics oriented, pre-condition oriented, and clustering oriented [S7, S17, S22].

Quality metrics oriented approach is used to predict and identify refactoring opportunities (*Extract Subclass, Extract Superclass, Extract Class, Move Method, Extract Method, Pull Up Method, Form Template Method, Parameterize Method*, and *Pull Up Constructor*) [S7, S17, S22]. Most metrics are related to coupling, cohesion, and distance (similarity) between code elements, and the studies described several distinct ways to calculate such metrics.

Pre-condition oriented approach is used to identify refactoring opportunities (*Move Method*), mainly related to *Feature Envy* and

Table 9
Tools and secondary studies related.

Tools	S2	S4	S9	S13	S17	S18	S19	S20	S30	S31	S33	S39	S40	# of references
CCFinder	✓		✓				✓	✓	✓	✓				6
PMD	✓		✓	✓		✓						✓	✓	6
Checkstyle	✓			✓		✓						✓	✓	5
DECOR	✓			✓		✓						✓	✓	5
CloneDr	✓		✓					✓		✓				4
Nicad	✓							✓		✓			✓	4
inFusion	✓					✓						✓	✓	4
inCode	✓			✓								✓	✓	4
iPlasma	✓			✓								✓	✓	4
StenchBlossom				✓		✓						✓	✓	4
JDeodorant	✓					✓						✓	✓	4
JCD	✓								✓	✓				3
ConQAT/CloneDetective			✓					✓					✓	3
HIST		✓		✓									✓	3
JSpIRIT	✓					✓							✓	3
CloneDigger	✓		✓											2
Duploc								✓		✓				2
CP-Miner							✓			✓				2
Jplag									✓	✓				2
DECKARD							✓	✓						2
Clever			✓				✓							2
Simian							✓	✓						2
SDMetrics	✓												✓	2
JCosmo				✓									✓	2
NosePrints	✓												✓	2
CBSDetector	✓												✓	2
Gendarme	✓													1
Clone Miner			✓											1
SYMake	✓													1
CloneBoard			✓											1
CPC			✓											1
CloneScape			✓											1
Shinobi			✓											1
JSync			✓											1
Cleman			✓											1
CeDAR			✓											1
CloneTracker							✓							1
Cyclone							✓							1
CloneInspector							✓							1
Columbus							✓							1
Datrix							✓							1
SimScan							✓							1
SmallDude							✓							1
iClones							✓							1
CLAN/Covet								✓		✓				1
ARIES											✓			1
JCodeCanine				✓										1
JDev				✓										1
JCCD													✓	1
SAME													✓	1
Borland Together													✓	1
SonarQube													✓	1
InsRefactor													✓	1
Wrangler	✓													1
CodeRush			✓											1
Weka				✓										1
CodeNose						✓								1
2D-DSL													✓	1
Prodetection													✓	1
P-EA													✓	1
BLOP													✓	1
History Miner													✓	1
BBT													✓	1
TACO													✓	1
SMURF													✓	1
EvolutionAnalyzer													✓	1
Van													✓	1
Understand													✓	1
Pascal Analyzer													✓	1
SCOOP/Organic													✓	1
CodeVizard													✓	1
IYC													✓	1
MuLATo													✓	1
SpIRIT													✓	1
JCodeOdor													✓	1
JSNOSE													✓	1
HULK													✓	1
Paprika													✓	1
Metrics													✓	1
SourceMiner													✓	1
TrueRefactor					✓									1
IntelliJ IDEA	✓													1
SACSEA				✓										1
Fluid Tool													✓	1
Teamscale													✓	1
ConQAT													✓	1
BSDT													✓	1
# of tools	20	1	15	12	1	8	13	8	3	8	1	8	48	

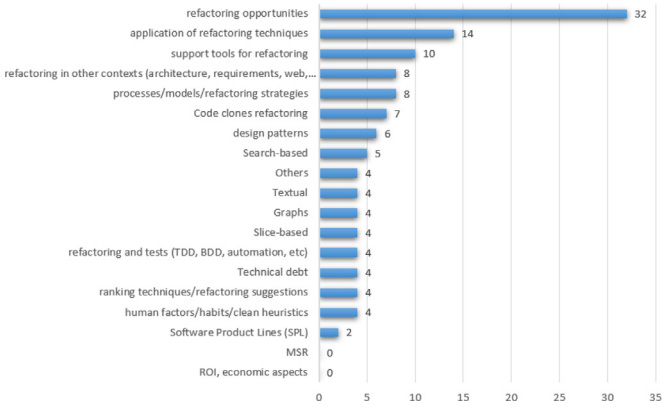

Fig. 6. Categorization of refactoring topics.

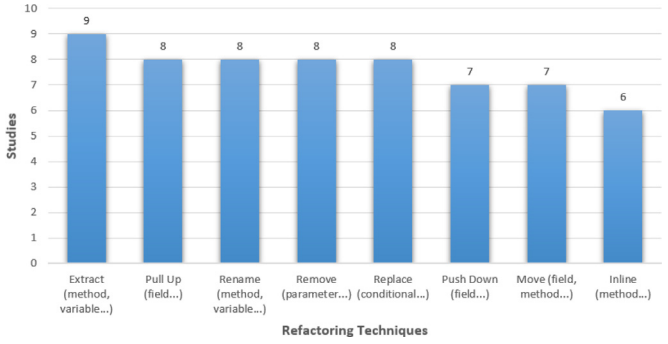

Fig. 7. Top 10 refactoring techniques.

Code Clones smells [S17, S22]. This approach firstly evaluates a condition just before applying a refactoring technique. Such a condition is also usually related to some metrics.

Clustering oriented approaches use algorithms based on some similarity measure and combination of code elements (*e.g.,* lines of code, attributes, methods, and classes), for refactoring opportunities (*Extract Method, Move Method, Move Class, Move Field, Inline Class*, and *Extract Class*) [S17, S22].

Graph-oriented approach, code slicing, and dynamic analysis are other approaches to the identification of refactoring opportunities (*e.g.,* *Move Method, Extract Method*, and *Extract Class*) [S22]. Such approaches are mainly useful to discover different refactoring opportunities in SPL [S14], and models [S3, S29, S36]. Too, code slicing is a practical approach to small code snippets (*e.g.,* methods), although having scalability issues [S22].

Moreover, the study [S22] relates that only a single approach for specific refactorings (Graph-oriented approach used to *Extract Interface* and clustering-oriented approach used to *Inline Class*).

Search-based approaches use to detect refactoring opportunities and to evaluate their applicability (application, behavior preservation, impact) [S17, S25]. The technique most used is the adoption of evolutionary algorithms (*Genetic Algorithms*), with highlights for *Hill-Climbing Search* [S17, S25]. The search-based approach has also explored this opportunity for pattern-oriented refactoring (Kerievsky, 2004), with studies of patterns *Template Method, Decorator, Abstract Factory*, and *Factory Method* [S17].

Some studies (see [S7, S22]) indicated that researchers generally compare the results obtained from one refactoring with the results of other refactoring that use the same identification approach. One of the key open issues in this area is analyzing the results of applying different approaches for identifying refactoring opportunities for a specific activity to determine the best approach. Indeed,

examining how refactoring techniques using different identification approaches can be further explored in future work.

Another important aspect of refactoring is how to apply it. The application of refactoring can *direct* or *indirect* [S25]. In the direct approach, the refactoring is applied directly to the artifact (*e.g.,* code and model), and thus it can be easily automated. In this case, the preservation of the behavior of refactoring is ensured. In the indirect approach, a sequence of refactoring produced as an optimized intermediary solution, and later that sequence is applied to the artifact. Thus, the artifact is indirectly optimized. Most of the work reported by [S25] uses **indirect approach and one possible reason may be the difficulty in ensuring the preservation of behavior**.

Automation is one of the critical difficulties in performing comparative studies among approaches. According to [S25], the refactoring process addresses six tasks [S10]. However, there is not a fully automatic approach for the whole software refactoring activity by solving all these tasks. Based on the results presented in [S25], the most difficult tasks are: (1) assure that the applied refactoring preserves behavior, (2) implement the refactoring; and (3) maintain the consistency between the refactored artifact. Still, according to [S25], **one of the problems of automating this task is the difficulty in preserving the behavior**.

In addition to automation, another interesting topic is the application of refactoring with tools support, discussed later in Sub-Section 4.3.

4.1.3. Impact on software quality

The studies [S4, S5, S7, S22] indicated that different refactoring sometimes have an opposite impact on different quality attributes. Performing unnecessary refactoring (changes in a code that does not need to refactored) may unexpectedly cause the code quality to degrade instead of being improved. Therefore, **refactoring does not always improve all software quality aspects**.

The studies [S7, S22] investigated the impacts of a few individual refactoring on some internal quality attributes such as cohesion, coupling, complexity, inheritance, and size. However, such studies were not able to identify impacts on external and other internal quality attributes.

Researchers were more interested in exploring the impacts of *Move Method, Extract Class*, and *Extract Method* on quality than the impact of any other refactoring [S7, S22]. Still, according to [S7, S13], researchers took two main approaches in studying the effect of refactoring on quality. The first approach is identifying refactoring opportunities, determining those required to remove code bad smells, performing refactoring when it is applicable, and comparing the code quality before and after refactoring. The second approach is analyzing the changes implemented on code during the maintenance phase, detecting the changes due to refactoring, and comparing before and after the code quality.

Each refactoring scenario includes: (1) a summary of the situation where refactoring is necessary, (2) a motivation for the importance of performing the required refactoring, and (3) a mechanism describing how to implement the refactoring. The study [S7] relates some refactoring and quality impact.

Extract Class was found to have a potentially positive impact on cohesion, inheritance, and size, and a potentially negative effect on complexity and coupling. *Extract Subclass* has a potentially negative impact on complexity and an inconsistent impact on cohesion and coupling. *Inline Class* has a potentially positive impact on cohesion, coupling, and complexity, but it has an opposite effect on inheritance. *Extract Method* has a potentially positive effect on cohesion, complexity, and size, and it does not affect inheritance and coupling (in most cases). *Move Method* has a potentially positive impact on cohesion and a potentially negative impact on coupling and complexity. *Move Field* has a potentially positive effect on cohesion and a potentially negative effect on the coupling. *Encapsulate Field* has a potentially positive impact on complexity, an inconsistent impact on coupling and cohesion, and does not affect inheritance. *Replace Data Value with Object* has potentially positive implications for cohesion and a potentially negative impact on the coupling. Finally, *Replace Method with Method Object* has a potentially positive impact on the coupling.

We observe such information about positive or negative impacts are very relevant to support developers in applying refactoring and assessing which refactoring used, not only to eliminate smells but also to improve quality aspects. Additionally, it is beneficial for **expanding studies on the impacts on quality in other refactoring, not yet explored**. In Section 5, we discuss more deeply the relationship between refactoring and quality.

4.1.4. Software evolution and technical debt

There is nowadays an agreement in SE community about technical debt: **refactoring are the primary approach to minimize the effects of technical debt** [S21, S26, S27]. Besides, if refactoring is overlooked, it can lead to a development crisis in the long run [S21, S27].

Decision-making about refactoring is a challenge because costs are concrete and immediate. In contrast, the benefits of refactoring are vague, long-term, and historically very difficult for the developers to quantify or justify [S21, S23]. The identification and application of refactoring may introduce new problems, and therefore, complicating the analysis.

One strategy to identify refactoring candidates [S21] is to locate the architecturally relevant classes as they are the pillar classes of the software design. To this end, we need finding classes that have earlier been frequently refactored together with looking for classes that are harmful to the system's design. The classes are prioritized and sorted according to their impact on the overall system's quality. However, as reported by [S21, S23, S26, S27], there is a lack of studies that conclusively describe the data caused by such code changes. Therefore, **architectural refactoring is risky, difficult to estimate, and very difficult to prioritize**.

RQ1 Summary

Challenges: The relationship between smells and refactoring is not a one to one relationship. It brings us numerous challenges regarding refactoring, such as (i) which refactoring can combine, (ii) which can not combine, (iii) which have the most significant impact on quality, and (iv) which detracts from the quality of software.

Although we have a large amount of research associated with refactoring, we still need to bring it closer to practice, encouraging researchers to improve the results most commonly used in practice. Another challenge is how to analyze the results obtained in the application of the refactoring.

Comparing how refactoring can use different identification approaches can be further explored in future work.

Observations: Extraction techniques are the most mentioned in the secondary studies. However, the number of techniques explored is still small (between 20 and 27 of 72). Refactoring opportunities, application of refactoring, and refactoring tools are topics that most appear in the studies. Quality metrics-oriented approach, precondition-oriented approach, and clustering-oriented approach are the most cited approaches to identify refactoring opportunities.

> Refactoring is the first approach to minimize technical debt effects. However, some refactoring, when applied, negatively affect the quality of the software.

4.2. RQ2: What smells-related topics have been investigated in secondary studies?

As noted in Fig. 5, most of the selected studies refer to the smells [S1, S2, S8, S9, S11, S12, S15, S16, S18, S19, S20, S23, S24, S26, S28, S29, S30, S31, S32, S33, S34, S35, S36, S37, S38, S39, S40]. In this section, we focused on design and code smells, because they are the most cited subjects in the selected studies and also due to their direct relationship with the code. The following are the main points.

4.2.1. Design smells highlights

Although studies primarily focus on code smells, papers about design smells are also found. Fourteen studies quoted design smells [S1, S2, S3, S13, S15, S19, S24, S28, S34, S35, S37, S38, S39, S40]. The top 5 design smells (Fig. 8) were defined by Brown et al. (1998).

Blob is the most mentioned design smell in the studies [S1, S3, S13, S24, S28, S34, S35, S37, S39, S40]. Some reasons can justify this mention. Blobs are easy to detect, and there are a variety of tools that identify this type of design smell. Table 8 presents a list of detection approaches and tools to detect Blob. Also, *Blob* is used in some studies as synonymous with *Large Class* (Fowler et al., 1999) or *God Class* (Riel, 1996). Previously, we presented the *Blob* definition by Brown (see Table 1) and the *Large Class* definition by Fowler (see Table 2). However, in other works, *e.g.,* [S34], this differentiation is made. Here, we have maintained this differentiation by considering the original definitions of distinct authors.

We can observe, in these cases, problems related to smells nomenclature. We also note that these naming and definition problems also occur with other smells. For example, *Copy and Paste Programming* has been used as a synonym for *Duplicated Code/Clones*. We were identifying distinct studies [S11, S12, S15, S39, S40] using different smells names to describe the same problem in design and code.

Some design smells usually appear together in many studies. For example, the following studies [S1, S3, S13, S24, S34, S39, S40] that discussed *Blob* also approached *Spaghetti Code, Swiss Army Knife, Lava Flow, Functional Decomposition*, and *Poltergeist*. Many works often consist of evaluating a given design smell or even its relationship with other design smells.

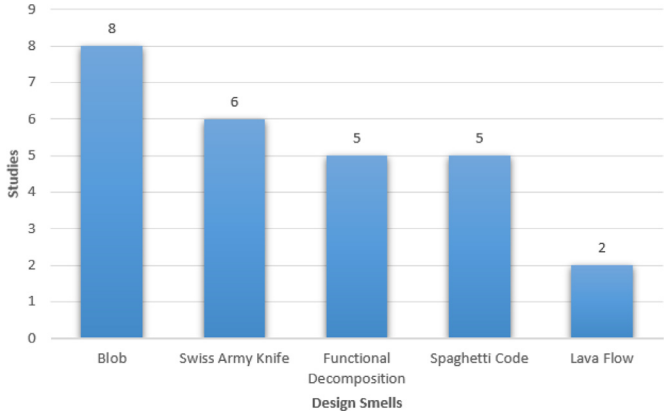

Fig. 8. Top 5 design smells.

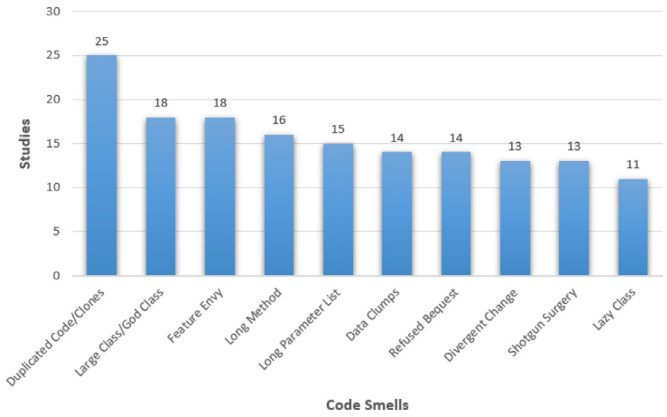

Fig. 9. Top 10 code smells.

The **design smells has also been studied together with code smells**. The studies [S28, S35, S37, S39] reported relations between the following pairs of design smells and code smells (*e.g., Blob, Data Class*, and *Blob, Large Class*). Others pairs of design smell we found are: (*God Class, God Method*), (*God Class, Feature Envy*), (*God Class, Data Class*), (*God Class, Duplicated Code*), (*Data Class, Data Clumps*), (*Divergent Change, Shotgun Surgery*), and (*Divergent Change, Shotgun Surgery, Feature Envy, Long Method*) are also related [S28, S39].

4.2.2. Code smells highlights

The code smells initially defined by Fowler et al. (1999) are the most mentioned. We presented top 10 most quoted code smells on studies (Fig. 9).

Twenty-eight studies quotes code smells [S1, S2, S3, S4, S8, S9, S11, S12, S13, S15, S16, S18, S19, S20, S23, S24, S27, S28, S30, S31, S32, S33, S34, S35, S37, S38, S39, S40]. *Duplicated Code/Clones* is the most studied code smell [S9, S19, S20, S30, S31, S32, S33] (7 studies from 28). We observe that the *Duplicated Code/Clones* have been investigated separately and explored in different ways. *Duplicated Code/Clones* cause code design problems, making maintenance difficult, and introducing subtle errors. Probably, It is the reason we find an active community dedicated to *Duplicated Code/Clones*.

Other code smells in the Top 10 most quoted list are *God Class/Large Class, Feature Envy, Long Method, Long Parameter List, Divergent Change, Data Clumps, Refused Bequest, Shotgun Surgery,* and *Lazy Class*.

When the technical debt was the subject, *God Class/Large Class* has been the most investigated smell. Such the smell is conceptually easy to understand, and, according to [S21], it is up to 13 times more likely to be affected by defects and up to 7 times more change-prone, which makes them a good candidate for TD mitigation. Several reasons justify the higher prevalence of some smells than others: tools available for their detection, the frequency of smell occurrence, popularity among practitioners, representativity of design and code problems, and the incidence of one code smell in another.

However, rarely some code smells are investigated. According to [S12, S18], *Alternative Classes with Different Interfaces, Incomplete Class Library* did not obtain the attention of researchers. The study [S12] still include *Primitive Obsession, Inappropriate Intimacy*, and *Comments*. Perhaps, these smells are not so interesting, or they are complicated to identify, not justifying the carrying out of studies. The literature does not explain why researchers did not attempt to detect them.

Some of the smells listed on the top 10 smells are related usually by co-occurrence. We observe in Fig. 10 the relationship among smells reported by studies [S1, S28, S39, S40]. The nodes

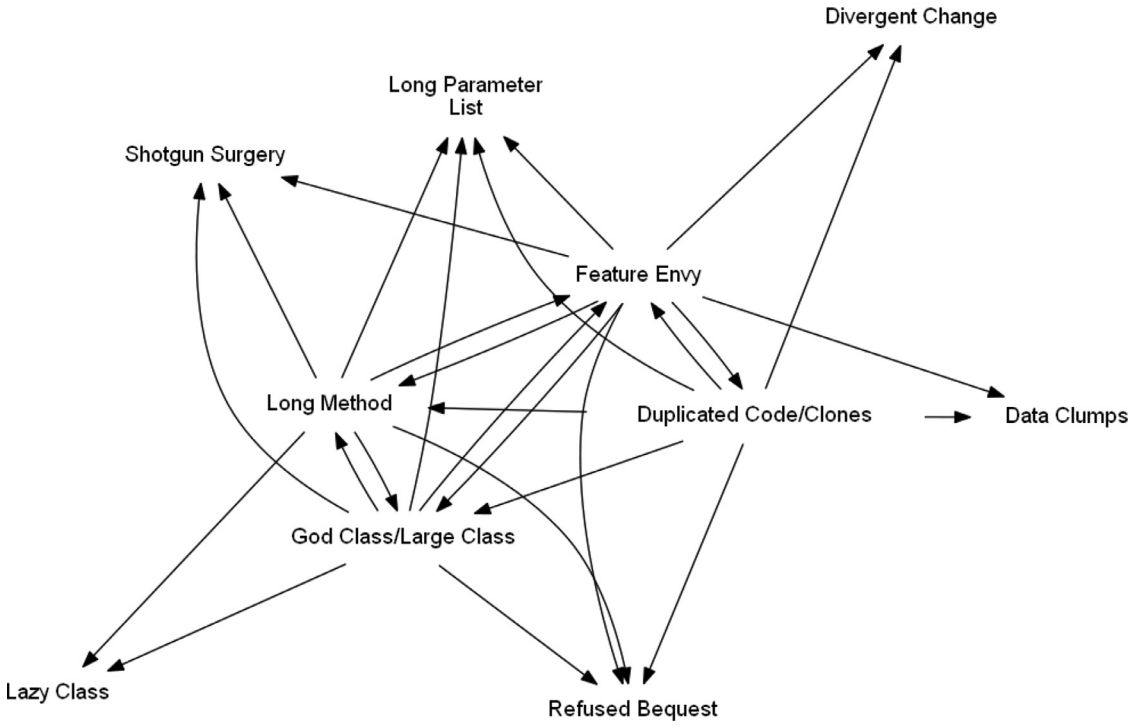

Fig. 10. Co-occurrence of smells from the studies.

represent smells, and the edges are the relationships between them.

We observed that the code smells *God Class/Large Class, Long Method, Feature Envy*, and *Duplicated Code/Clones* co-occur in many selected studies. For example, the presence of a *Long Parameter List* can result in a *Long Method*. The presence of *Long Method*, by its characteristic, can indicate a *God Class/Large Class*. Also, the fact that we separate a *Long Method* and it has many behaviors that are not related to the same class, can cause a *Feature Envy*.

Other code smells, like *Lazy Class, Refused Bequest, Shotgun Surgery, Long Parameter List, Divergent Change*, and *Data Clumps* are mentioned in studies, but the relation between them is not mentioned, suggesting that this is still a topic deserving more attention. The current studies on the co-existence of smells in the code indicate an association with maintenance and design problems. **Co-occurrences can be more explored, such as the appearance of smell in consequence of another smell, smells that are always close (presence of one implies the presence of another), among others.**

In the same way, it occurs with design smells, naming problems are also found with code smells. Several studies [S11, S12, S15, S39, S40] claim that the **use of terms and classifications adopted by different authors are not sound**. On the one hand, we found distinct definitions for the same smell name. On the other hand, the same smell definition is presented with different names. According to [S40], such fragmentation of definitions is due to the lack of a more systematic or formal taxonomy for code anomalies.

The standardization process is necessary to allow the unification of the terminology and its precise definition. A standard, cataloging all the smells (design/code) defined up to the present time should be possible, determining those that refer to the same smell with different names. The study [S39] also suggests the creation of a unique catalog (in the same way as the *Design Patterns Catalog*) with a unique entry in the catalog enriched with *"other names"* or *"also known as"*. It is **important to increase efforts to the standardization of the concepts, which would also allow an increase in smells detection consistency**.

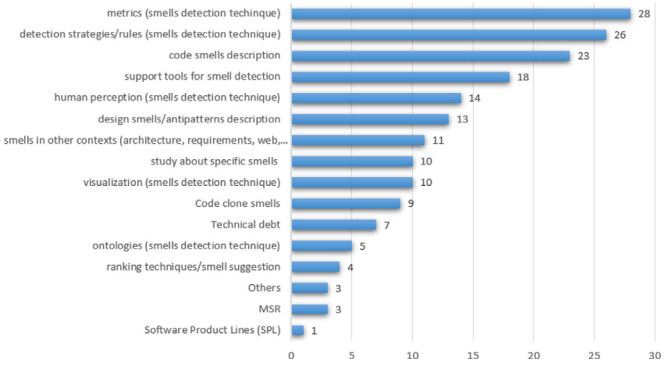

Fig. 11. Smell topics, based on how are discussed in the literature.

4.2.3. Smell detection approaches

The main topics for code smells are shown in (Fig. 11). The most discussed topics are smell detection approaches, appearing in six topics among the 16 most cited topics. There is a lack of standard agreed-upon definitions for code smell detection in the research community [S2, S11, S12, S39, S40].

We analyze the main approaches to smell detection, considering the top 10 most quoted smells (Table 8). Among them, the use of human perception, metrics-based, detection rules, reverse engineering/static analysis, history-based, machine learning-based, and software visualization are the most mentioned approaches. We group them to facilitate our analysis. We defined the following classifications: human perception, metrics-based, strategy/rules-based, probabilistic/search-based, and visualization-based.

Human perception-based approach [S10, S12, S13, S15, S18, S19, S20, S24, S26, S34, S35, S38, S39, S40] is a fundamental manual approach to detect smells, usually based on different guidelines followed by developers to detect manually design defects [S18, S39]. Manual techniques are human-centric, time-consuming, and error-prone. These techniques eliminate uncertainties in the detection

process due to human involvement, but they are not useful for examining code smells within large systems. However, we do not eliminate human participation from the detection process.

Metrics-based approach [S2, S4, S5, S6, S7, S9, S12, S13, S14, S15, S16, S17, S18, S19, S20, S22, S24, S25, S26, S30, S31, S32, S33, S34, S35, S38, S39, S40] is usually the approach used to detect *Large Class/God Class, Long Method, Data Clumps, Refused Bequest, Shotgun Surgery* and *Lazy Class*. This approach was mentioned in all top 10 smells selected. It used to evaluate/measure source code elements (*e.g.,* attributes, lines, parameters, methods, classes), allowing them to take some decisions. The accuracy of metrics-based approaches is dependent on the proper selection of threshold values, which are usually empirical and not much reliable [S2, S18, S38, S39]. There is not yet a **consensus on the standard threshold values for the detection of smells, and consequently, there is a lot of disparity among results of different techniques**. One of the factors that can contribute to this finding is the lack of standardization/formalization of the smell definition.

Rule-based (or strategy-based) approach [S2, S4, S6, S9, S11, S12, S13, S14, S15, S17, S18, S19, S20, S22, S24, S25, S26, S30, S31, S32, S33, S34, S35, S38, S39, S40] is another approach which combines rules, logic expression, and metrics used typically to detect the following smells: *Feature Envy, Long Parameter List*, and *Divergent Change*. Different smells are represented as detection rules. Each rule is specific to specific smells and can be defined manually or automatically using different techniques [S39]. The conversion of symptoms into detection rules requires analysis and interpretation effort to select the proper threshold values. There is not yet agreement on defining standard symptoms with the same interpretations, and thus the precision of the approach is low. Since rule-based approach makes intensive use of metrics, the same works that mention this type of approach also mention a metrics-based approach.

Probabilistic/search-based approaches [S12, S15, S18, S20, S31, S32, S33, S34, S39] apply different algorithms and rules for the detection of smells directly from source code. Most techniques in this category apply machine learning algorithms and fuzzy logic. The study [S39] reinforces the use of the search-based approach to detect different types of smells. Several techniques and algorithms proposed for extracting specified rules to detect smells with techniques based on genetic and heuristic search algorithms. These techniques learn from the standard design and coding practices and examine how the code deviates from these practices. The success of these techniques depends on the dataset's quality and training [S2]. These techniques are very limited for dealing with unknown and varying definitions of code smells, but it is one of the approaches to be more explored in future work, not only for smell detection but also to support refactoring recommendations.

Visualization-based approach [S9, S12, S13, S15, S18, S20, S26, S34, S39, S40] integrate the capability of human expertise with the automated detection process. In some cases, when the software is very complicated, the graphical representation of the software artifact arises as a solution to deal with complexity. Such an approach has scalability problems for large systems, and it is error-prone because of wrong human judgment depending on visualization type. However, this approach could help developers to identify points of code to be improved, reducing the technical debt.

According to [S2, S39, S40], **smell detection approaches and their corresponding produced results are highly inconsistent. In general, generic approaches are used for all types of smells, while some specific approaches are used for more specific smells**. That is the case for approaches such as history-based, optimization-based, and probabilistic/search-based. The study [S40] reports that different detectors for the same smell produce different answers, which is coherent with the need for new strategies to identify smells (or even approaches) in a more efficient/effective way than current approaches. **It is also necessary to explore whether the approaches could be combined or individually used for the detection of a set of smells**. We suggest, as future work, the assessment of which approach combination is better and in which context and conditions.

4.2.4. Impacts and effects

The studies [S11, S15, S24] do not differentiate between a smell and a definite quality problem. The community believes existing smell detection methods suffer from high false-positive rates. Also, existing methods cannot define, specify, and capture the context of a smell adequately. Undoubtedly, the **impact of smells causes decay in the overall design, affecting the quality attributes** [S24, S35, S37, S38], including maintainability (the effort to change the code), understandability, and extendability (the effort to add new functionality). A deeper discussion of the relationship among quality attributes, smells and refactoring is presented in Section 5.

However, some studies [S15, S23, S24] do not establish an explicit connection between smells and their impact on the productivity of a software development team. Several studies [S21, S23, S26, S27, S38, S39, S40] presents different factors in how smells affect the architecture decay, both developer-focused and development process-focused, concentrating on the relation between design/code smells. Developer-focused issues involve difficulties related to inexperienced/novice developers focused on functionality build, lack of a systems architecture knowledge, apprehension due to system complexity, and carelessness. Development process-focused issues include difficulties related to missing functionalities, violation of object-oriented concepts (abstraction, information hiding, modularity, and hierarchy), project deadline pressures, changing and adding new requirements, updating new software and hardware components, and ad-hoc modifications without documentation.

The studies suggested that developers should promptly identify and address the code smells upfront. Otherwise, code anomalies increase modularity violations and cause architecture degradation. To achieve such skills, introducing new approaches to developers' education could be necessary. We do not find secondary studies discussing ways to teach such practices while developers are coding. Therefore, we suggest the development of mechanisms and tools which help developers, recommending practices in such context, as seen at Section 6.

RQ2 Summary

Challenges: The literature does not explain why researchers did not attempt to detect *Alternative Classes with Different Interfaces, Incomplete Class Library, Primitive Obsession, Inappropriate Intimacy*, and *Comments*. It is necessary to evaluate the reason for the lack of interest in these smells.

A smells naming standardization is necessary, allowing the terminology and its precise meaning to be unified. With this standardization, cataloging the smells defined up to the present time should be possible, determining those that refer to the same smell with different names. This process will undoubtedly have repercussions on detection approaches. Another question that can investigate is the appearance of smell in consequence of another or the existence of smells that are always related, sometimes co-occurring.

Furthermore, it is necessary to explore which approaches can be complementary or explicitly used for a specific smell. There is a variety of approaches to reveal-

ing smells, with high false-positive rates. Thus, there are open possibilities to explore and to improve methods capable of defining, specify, and capture the smell context.

Observations: *Blob* and *Duplicated Code/Clones* are the most mentioned design and code smell, respectively. Related to technical debt, *God Class/Large Class* has been the most investigated smell. Design smells have also been studied together with code smells. There are simple and composite smells (the combination of simple smell can lead to a composite smell).

It is a consensus that manual detection is difficult, time-consuming, and prone to errors. However, there is no consensus on the standard threshold values for the detection of smells, which are the cause of a disparity in the results of different techniques.

Over time, there has been a significant number of detection approaches, like metrics-based and strategies/rules. They have been the most cited. Other approaches, such as history-based, optimization-based, probabilistic-search-based, and visualization, have also been used to smells detection. Besides, smell detection approaches and the corresponding produced results are highly inconsistent. The impact of smells causes decay in the overall design, affecting quality attributes.

4.3. RQ3: Which tools have been mentioned for code-smell detection and refactoring support?

Manual detection of code smells in the early days was very time consuming, error-prone, and costly. Smell detection tools automate specific smell detection techniques. To address these problems, researchers developed many semi-automated and automated code smell detection tools and refactoring support. To answer RQ3, we investigated which platforms/programming languages are used and which tools have been aiming for smell detection. Furthermore, we also investigated supporting tools for refactoring.

The tools investigated in studies [S2, S13, S18, S39, S40] have many characteristics, summarized as follows: if they are free or not; whether they are open source or proprietary; their supported languages; the terms used to describe the smells; the internal representation of the software artifact; the degree of automation; the ability to also perform refactoring; the way to run the tool; their ability to generate metrics; the type of input source; the output format; the facility to work with *Command Line Interface* (CLI); *Graphical User Interface* (GUI)/plugged on IDEs, and the list of smells the tool can detect.

4.3.1. Platforms/programming languages

The majority of studies [S1, S2, S3, S4, S5, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S22, S23, S24, S25, S26, S29, S30, S31, S32, S33, S34, S35, S36, S37, S39, S40] refer to **Java as platform/programming language(PL) most used to develop tools** (Fig. 12).

Thirty-five studies that mentioned platforms/PLs have suggested Java as the most used technology, following by C++ and C# (57.14% and 42.85%, respectively). According to GitHub,[3] Java is the third most popular programming language used in project repositories, whereas according to the Tiobe Index,[4] Java is the most popular programming language. Java has been the most widely used platform for tool development and also as a target language for experiments. Few tools (*e.g., PMD, Borland Together, CCFinder, inFusion, inCode*, and *iPlasma*) listed in the studies [S2, S18, S39, S40] work with more than one programming language. Therefore, **there is an opportunity to develop tools that support more than one programming language**. For instance, Javascript is more and more adopted by the industry and is already the leader in FLOSS projects on GitHub. However, it appears only as of the fourth technology mentioned in the studies so far. Nevertheless, the second edition of the recently released Fowler refactoring book (Fowler and Beck, 2019) presents all examples in Javascript.

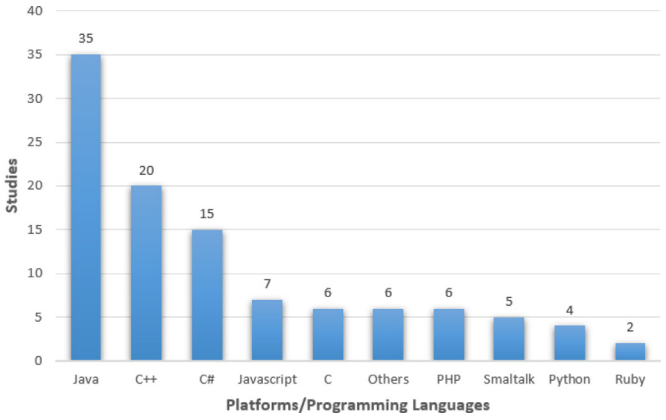

Fig. 12. Platforms/programming languages more used.

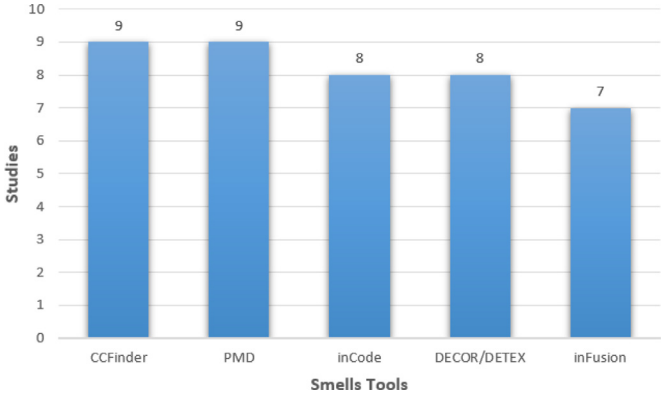

Fig. 13. Top 5 smells tools.

4.3.2. Smells detection tools

Nineteen studies [S1, S2, S9, S12, S13, S17, S18, S19, S20, S26, S27, S30, S31, S33, S34, S35, S38, S39, S40] quoted smell detection tools. **In a set of more than 162 distinct tools, we presented the top 5 smell detection tools** (Fig. 13). *CCFinder* is the tool that most appears in studies [S2, S9, S19, S20, S30, S31, S34, S35, S39] for smells detection, together with *PMD* [S1, S2, S9, S12, S13, S18, S26, S39, S40] (both with 47.36% of studies). *inCode* [S1, S2, S13, S18, S34, S35, S39, S40] and *DECOR/DETEX* [S13, S18, S34, S35, S38, S39, S40] appears together with 42.10%. *inFusion* [S2, S13, S17, S18, S35, S39, S40] appears with 36.84%.

CCFinder is the most quoted tool to detect *Code Clones* (*e.g., Type-1* and *Type-2*), using the token-based approach. The tool uses a suffix tree algorithm, and so it cannot handle statement insertions and deletions in *Code Clones*. It applies several metrics to detect relevant clones. It also optimizes the sizes of programs to reduce the complexity of the token matching algorithm. It produces high recall, whereas its precision is lower when compared with some other techniques [S31]. *CCFinder* probably is the most cited because detects *Type-1* and *Type-2* clones, which are the more

[3] More info: https://octoverse.github.com/, accessed December, 09 2019.
[4] More info: https://www.tiobe.com/tiobe-index/, accessed December, 09 2019.

commons clones. There is not a "perfect" clone detection technique, e.g., having high scores to all properties like precision, recall, ok, portability, and robustness [S9, S31]. Perhaps new clone detection approaches could do better by overcoming some of the limitations of existing techniques. Still according to [S31], ***Type-4*clones require to solve an undecidable problem**. The clone detection technique can be improved by combining several different types of methods or re-implementing systems using a different programming language. Such the technique presents new challenges for software maintenance, refactoring, and clone management. **Clones introduce maintenance and evolution problems, but, in most of the cases, they do not affect quality** [S20, S35, S37].

PMD is a static analysis tool used to detect code violations or bad development practices. Because of its broad action spectrum, it ends up detecting some smells like *Duplicated Code, Large Class, Long Method*, and *Long Parameter List*. For *Large Class* detection, the tool presents a low precision rate (about 14%). On the other hand, it performed well with *Long Method*, achieving 50% to 67% of recall and 80% to 100% of precision [S2]. In the study [S18] is discussed the use of *PMD*, comparing with another coding standard/smell detection tool, called *Checkstyle*.[5] Although *Checkstyle* detects the same smells of *PMD*, there is a difference in detection results, due to different threshold values used by these tools, as well as the metrics took in the account. Beyond the list of smells tools, the study [S18] presents a set of experiments realized. For *Large Class, PMD* uses a threshold of 1000 NLOC,[6] while *Checkstyle* uses 2000. Still, for *Long Method, PMD* uses a threshold of 100 and *Checkstyle*, 150. For *Long Parameter List*, again we have differences of thresholds, with *PMD* using ten and *Checkstyle*, 7. PMD appears among the most cited due to its more general performance, although, as we have seen, it is not very accurate.

inCode is an Eclipse plug-in that helps smell detection, detecting *Duplicated Code, Large Class, Feature Envy, Data Clumps*, and *Refused Bequest*, with visualization support. Although it is a tool that detects several smells [S2, S35, S39, S40], we do not found secondary studies discussing more details about functionalities, strategies used to detect, precision, and recall of operation. Since *inCode* is an Eclipse plugin detects multiple smells and works with multiple programming languages, it is among the most commonly cited detection tools.

DECOR/DETEX, proposed by Moha et al. (2010), **is the tool capable of detecting more than 10 smells** (*e.g., Large Class/God Class, Lazy Class, Long Method, Long Parameter List, Refused Bequest, Speculative Generality, Message Chains, Shotgun Surgery, Duplicated Code, Comments, Data Class*) identified by Fowler et al. (1999). Also, *DECOR/DETEX* detects design smells proposed by Brown et al. (1998), like *Swiss Army Knife, Blob*, and *Functional Decomposition. DECOR* is a method, and *DETEX* is a tool that allows us to specify and to detect code and design smells, using a DSL[7] [S13]. The tool allows developers to make metrics threshold settings to detect smells. Although the study [S18] reports 50% of precision and 100% of recall, the study [S13] reports experiments indicating not such a high accuracy. *DECOR* is among the most cited detection tools probably due to the lightweight nature that allows researchers to employ it to detect several types of smells without the need of compiling anything each time.

inFusion is another tool that detects *Duplicated Code, Large Class, Feature Envy, Long Method, Data Clumps*, and *Refused Bequest. inFusion* has an open-source version called *iPlasma*, which is quoted too, but with more limited functionalities. The tool presents the same numbers of *PMD* [S2], with a 14% recall rate for *Large Class* detection. The authors report an experiment with just one project. Probably, it is necessary to achieve more experiments. Also, the tool performed well with *Long Method*, achieving 50% to 67% of recall and 80% to 100% of precision.

Smell detection tools use thresholds on metrics or ad-hoc rules to identify structures in code, at the price of some inaccuracy [S40]. The accuracy of a code smell detection tool is a key aspect of its validity. Some secondary **studies [S2, S15, S18, S40] reveal approximately 30% of the tools spotlight the accuracy, that is, precision and recall, of their technique or tool**. Also, the studies [S18, S40] describe that authors of code smell detection tools perform experiments on different systems, and the comparison of published results becomes difficult when tools are not available. **Standard benchmark systems for code smell detection tools are not available, which require the attention of the research community.**

Murphy-Hill et al. (2012) presents a list of guidelines as success factors related to usability for code smell detectors. The studies [S2, S40] contain a discussion about usability and detection tools. They define six features for tools analysis: (a) easy exportation: results about the detected bad smells were easily exportable, for instance, to text, CSV or other file formats; (b) highlighting of smell occurrences; (c) configurability: allowing detection settings; (d) graph visualization; (e) detected smell filtering; (f) Analysis of multiple versions. According to [S2], *inFusion* is the only tool that supports five features (*a* to *e*), although two of these are available only in the full commercial version of the tool. In addition to the features mentioned above, the studies [S2, S40] describe that some usability issues could hinder the tool user experience. Some usability problems, such as difficulty in navigating between bad smell occurrences (in general, results showed in long lists without summarization), difficulty in identifying the source code related to a smell detection, and lack of advanced filters for specific bad smell detection. In general, tools do not provide data visualization through statistical analysis, counters of detection results, or result's presentation by charts.

Most of the tools focus on the recovery of code smells from a single language, that is, in most cases, Java language [S2, S18, S39, S40]. None of the tools detect all 22 code smells identified by Fowler et al. (1999). On average, tools cover three to four smells for detection [S2, S18]. The tool *inFusion* claims to detect all code smells of Fowler, but it is a commercial tool, and it was not free and available for experiments realized in [S18, S39]. We identify an opportunity for the development of tools for the detection of relevant smells not yet explored. Additionally, tools that detect smells in more than one language should also receive attention from the SE community.

4.3.3. Refactoring tools

Since refactoring tools are also essential to our work, and we did a study on tools that support refactoring. Thirteen secondary studies [S1, S2, S4, S9, S12, S13, S14, S17, S18, S29, S34, S39, S40] **presented 24 distinct tools that help developers applying refactoring. We select the top 5 refactoring tools** (Fig. 14) for a detailed discussion.

***JDeodorant* is the tool that most appears in studies** [S1, S2, S4, S12, S13, S18, S29, S34, S39, S40] (71.42% of studies), followed by *TrueRefactor* [S2, S17, S25] (21.42%), *Eclipse Refactoring* [S18, S29] and *IntelliJ IDEA Refactoring* [S2, S29] (14.28% both), and finally *Wrangler* [S13] (7.14%). There are other 19 tools with a similar percentage, but we choice *Wrangler*, because it is the first tool supporting refactoring for clones.

JDeodorant [S13, S18, S40] is an Eclipse plug-in that automatically recognizes *Large/God class, Feature Envy, Switch Statement/Type Check*, and *Long Method* code smells from Java source code. It has

[5] *Checkstyle* was the sixth tool most mentioned in the studies.
[6] non-commented lines of code.
[7] *Domain Specific Language.*

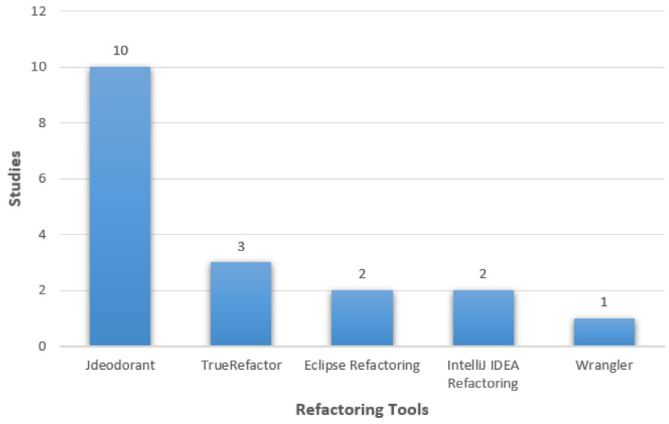

Fig. 14. Top 5 refactoring tools.

support for refactoring and assists the user in refactoring transformations [S18, S40].

In the same study [S2] that also analyze *PMD* and *inFusion*, there was high agreement among these three tools concerning the detection results, although *JDeodorant* points out more bad smell instances compared to the other tools, in its default configuration. *JDeodorant* has too achieved low precision rate (about 14%) in detecting *Large Class*. The tool uses both metrics and AST[8] to detect bad smells. Considering that some tools apply unknown techniques, detection results may be different. The authors [S2] observe that *JDeodorant* indicates the highest number of *Large Class* and *Long Method* instances, and scored the lowest results for both recall and precision. We note that *JDeodorant* is one of the few tools that combine smells detection with the automatic application of refactoring. It may be the reason for being the most mentioned: even with some limitations presented, this shows that more tools with these characteristics are needed.

The *TrueRefactor* [S17, S25] is an automated refactoring tool that significantly improves the comprehensibility of legacy systems (Griffith et al., 2011). To detect code smells, each source file is parsed and then used to create a control flow graph to represent the structure of the software. For each code smell type, a set of metrics is calculated to identify whether a section of the code is an instance of a code smell type. The tool uses a genetic algorithm (GA) to search for the best sequence of refactoring that removes the highest number of code smells from the source code. As an automated refactoring tool, *TrueRefactor* does perform actual refactoring, but currently supports mainly the modification of UML rather than code. The study [S17] describes an example program with code smells artificially inserted to analyze the effectiveness of the tool. The number of code smells of each type over the set of iterations was measured jointly with the measure of a set of quality metrics. In both cases, the values increased initially before staying relatively stable throughout the rest of the process. A comparison of initial and final code smells shows that the tool removes a significative proportion of smells, and also metric values indicate that the surrogate metrics are improved. Despite the limitations presented by *TrueRefactor*, a positive aspect is a way adopted for sorting the most appropriate refactoring, based on a given smell and its impact. This shows that researchers can advance in more studies of classifying refactoring, taking into account their impact.

Beyond the *JDeodorant* and *TrueRefactor*, the rest of the tools do not present more details or analysis about use and accuracy. Two IDEs appear, too, as most mentioned. The *IntelliJ IDEA* implements more than 40 refactoring, using a lexical and syntactic parser to convert the code into the form of AST, called *Program Source Interface* (PSI) (Jemerov, 2008). The PSI is used to validate any generated code. After code transformation, the *Formatter* is responsible for verifying the scope of the changes, adjusting the code with indentation, inserting blank lines, changing of qualified names, and imports of libraries. *IntelliJ IDEA* still makes use of a built-in DSL to find fragments in the PSI using a defined lean notation. The use of DSL is also one of the paths suggested for future studies for code refactoring (Zhang et al., 2011; Jemerov, 2008; Li and Thompson, 2012).

Wrangler (Li et al., 2006; 2008) is a tool that supports interactive refactoring of Erlang programs. It is integrated with Emacs as well as with Eclipse, through the ErlIDE plugin. Wrangler itself is implemented in Erlang. The tool supports a variety of refactoring, as well as a set of code smell inspection functionalities, and mainly a lot of facilities to detect and eliminate *code clones*.

Eclipse Refactoring is also well-known for the constant improvements to the use of refactoring. The process consists of the phases of verification of preconditions, detailed analysis, and rewriting of the code. Guidelines help to seek a more straightforward code rewriting mechanism, also based on the AST form. Although *Eclipse* supports more than 20 refactoring techniques, a lot of work has improved the use and application of the refactoring, resulting in more speed for developers (Murphy-Hill and Black, 2007b).

We note that *Eclipse* and *IntelliJ IDEA* appear here by bringing automation to refactoring, helping developers in the process. One of the advantages of using these tools is to ensure the application of refactoring (passed by refactoring preconditions and also by postconditions, ensuring that AST has not been broken). However, it is up to the developers to find the smells and also know the refactoring to apply. Murphy-Hill has argued in previous work (Murphy-Hill and Black, 2008; 2010) on usability and habits of developers, showing that this is not a trivial process. We think tools taking advantage of such potentialities, guiding the developers in the process (for example, recommending a refactoring), can be explored in future studies.

4.3.4. Tool considerations

According to [S13, S18, S21], the **detection of code smell reduces the cost of maintenance if the failures found in the early stages of software development**. The applicability of smell detection tools varies according to the goal of detection: the objective could be software quality management, or maintenance after smells detection, code quality improvement, and fault detection via refactoring. One observation is the detection goal is highly related to the impact goal, since the investigation of the impact of smells occurs after the detection. Indeed, the study [S40] observes that the inconclusive knowledge about the negative impact of smells is partially attributed to tools/techniques used to detect them.

Since there is a great variety of tools and discrepancies on tools findings, we cannot discard discrepant results in different studies because they were using different tools. Still, according to [S39], only a few tools can analyze very large-sized projects (millions of lines of code). Most of the tools did not take into account the expert feedback or other characteristics like the influence of the context, project domain, and project status.

A **large number of tools are available for the detection and removal of code smells. However, evaluation frameworks that could help the user for appropriate selection of any tool for a given context are missing** [S18, S39]. The **currently available tools** [S2, S15, S39] **can detect only a very small number of smells**. It is still a big problem to determine which code smell is effective in indicating the need for refactoring and what type of refactoring, and programmer involvement is still necessary [S2, S11]. Although refactoring has been proposed to remove smells, several subtleties make this activity inherently complicated [S40].

[8] *Abstract Syntax Tree.*

In the refactoring field, we suggest that developers of new detection tools should be aware of the possible usages of their tools, considering observations referenced by [S2]. According to [S5], **a refactoring tool developer can not provide custom refactoring fitting for all specific user needs because the possible number of refactoring is unlimited**. Therefore, customizable refactoring tools based on the demand of the developer are missing. Still, as reported by [S7], the existing refactoring tools are error-prone, and therefore, using these tools may result in producing incorrectly refactored pieces of code. As a result, tools usage sometimes negatively affect code quality. Automating the refactoring process consists of automating the two following main steps [S22]: (1) identify refactoring opportunities, and (2) perform refactoring. It is necessary to automate effective techniques to identify opportunities for such refactoring and then perform it.

Tools should make it easier for programmers to refactor quickly and correctly. **Tools have to help analyze the impact of the smell: nowadays, many tools do a weak job of communicating errors triggered by the refactoring**.

Furthermore, tools like *Eclipse IDE* and *IntelliJ IDEA* automate some pointed refactoring. The point is it depends on the developer knowing how to conduct it. In particular, the study of the human perception of what is a code smell and how to deal with it has been mostly neglected in the past [S15]. In the same way, human opinion remains important to decide where refactoring is worth applying [S5]. According to [S18, S39], there are few studies discussing such support for refactoring. A significant motivation for identifying code smells is source code refactoring, but most code smell detection tools focus on the detection/visualization of code smells. This finding is evidenced in our research (Table 8, marked as *): there is not a tool that automates refactoring, according to one indicated smell. It shows the need to deepen work on the constructions of refactoring support tools coupled to smell detection tools.

Moreover, from the perspective of tools evaluation, the unavailability of implementations hinder reproducibility and impose barriers to the underlying empirical studies, in particular for those aiming at comparing new approaches with the state-of-the-art. **Experts from industry and academy need to assess the results of the tools regarding detecting false positives and false negatives**. It is also essential to have expert opinions concerning smell prioritization, smell impact on product quality and technical debt, as well as evaluation of refactoring in the same terms. In general, according to [S18, S39], tools **lack maturity and a lot of limitations restrict their use and adoption by the industry**.

RQ3 Summary

Challenges: We notice a preference for a programming language/platform, for both the tool construction and realization of experiments. It opens the opportunity to develop tools that support more than one programming language.

It is necessary to expand the studies and experiments, evidencing the accuracy of the smell detection tools as well as refactoring tools.

We identify a lack of maturity of tools with limitations that restrict their use and adoption by the industry. Standard benchmark systems for results of code smell detection tools and refactoring tools are not available, which require the attention of the research community. Also, it is necessary to involve experts to assess the results of these tools.

Observations: Several aspects characterize tools: free or not, open-source or proprietary, supported languages, the terms used to describe the smells, the degree of automation, the ability also to perform refactoring, the way to run the tool, among others.

Java is the platform/programming language most used to develop tools. Also, most tools focus on the identification of code smells from a single language, where Java is predominant too.

We have a large number of smell detection tools, using the most different detection approaches. Currently, available tools can detect only a tiny amount of smells (between 3 and 4). The number of tools that perform refactoring is small.

The most quoted smell detection tool is *CCFinder* because we have more studies related to the *Duplicated Code/ Clones*. This smell has an impact on software maintenance and evolution, but we have not identified significant effects on quality. The most cited refactoring tool is the *JDeodorant*, used to apply specific refactoring for specific code smells.

4.4. RQ4: Which RQs have been studied on smells and refactoring? what are the highest cited secondary studies?

Responding to RQ4, we analyze which RQs discussed in the studies and the correlation among them. Also, we present a discussion of the most mentioned works.

4.4.1. Analysis of RQs

RQs are essential points in any scientific work. They define the direction and conduct the focus of the studies. An explicitly stated RQ is one of the requisites for a review to be considered systematic. We explore RQs from two points of view: general analysis and specific analysis.

In a more general analysis, **we had 181 RQs in the 40 selected secondary studies**. It gives an average of 4.5 RQs per study. **The study [S40] was the study with the highest number (13 RQs)**. Thus, it is a broad study, addressing several topics related mainly to smell definition, smells detection, tools, impact, trends, and focus on the research group and researchers.

We had two studies [S28, S30] with only one RQ. The study [S28] addresses the co-occurrence between smells and design patterns. The study [S30] is focused on *Code Clones*.

Other studies, such as [S4, S9, S10, S18, S31], had no explicitly defined RQs. The studies [S4, S18] are SLRs. They have defined goals, but not in the format of RQs, not following a SLR protocol and making difficult an analysis. The studies [S9, S10, S31] are Surveys. Surveys do not explicitly have RQs and do not follow a defined protocol, unlike SLRs and SMs.

In a specific analysis, we used the RQ classification proposed by Easterbrook et al. (2008) (Table 10). This classification has been used in other studies as well, *e.g.,* da Silva et al. (2010, 2011), Petersen et al. (2015), Garousi and Mäntylä (2016). We ranked each RQ within the study, but we did not rank the studies based on their RQs, using the most specific RQ as the basis, as performed in da Silva et al. (2010). We observed that the studies [S1, S14, S16, S21, S27, S40] have one major RQ is divided into secondary RQs. In these cases, we classified the secondaries RQs follow the Easterbrook RQ classification. For brevity and space constraints, we are not reporting the entire list of RQs extracted from all studies, but the reader can find it in our online replication package (Lacerda, 2019).

Table 10
Classification of RQs, as presented by Easterbrook et al. (2008), number of studies, number of RQs in the pool and examples.

RQ Category	Sub-Category	Code	# of studies	# RQs in the pool	Examples
Exploratory	Existence	E	13 *(32.5%)*	16 *(8.83%)*	*Does X exist?* RQ1: What is the definition of a software smell? [S15] RQ3.3: Are there any analysis methods for detecting and/or evaluating ATD? [S21] RQ1.1: Are there bad smells significantly more studied than others? If so, is there any specific reason? Are bad smells studied alone or together with other bad smells (co-occurrences)? [S40]
	Description and Classification	DC	30 *(75%)*	90 *(49.72%)*	*What is X like?* RQ2.2: What co-occurrences have been identified by the studies? [S1] RQ2: Which are the main features of these tools? [S2] RQ3: What methods have been used to study Code Bad Smells? S[11]
	Description-Comparative	DCO	7 *(17.5%)*	11 *(6.07%)*	*How does X differ from Y?* RQ2: how to compare refactoring tools and techniques? [S6] RQ2.1: What are different studies in semantic clone detection and their comparative analysis? [S20] RQ1: What are the types of technical debt and what is not considered as technical debt? [S26]
Base-rate	Frequency Distribution	FD	11 *(27.5%)*	19 *(10.49%)*	*How often does X occur?* RQ3: Which are the most frequent types of bad smells these tools aim to detect? [S2] RQ1: What refactoring scenarios were accounted for in the PSs? [S7] RQ1: How many papers were published per year? [S17]
	Descriptive-Process	DP	10 *(25%)*	16 *(8.83%)*	*How does X normally work?* RQ2: What model smell detection strategies have been used to identify refactoring opportunities for model refactoring? [S3] RQ2: What are the different approaches used for the detection of code smells and how the smells are removed using these approaches? [S13] RQ4: How do smells get detected? [S15]
Relationship	Relationship	R	12 *(30%)*	13 *(7.18%)*	*Are X and Y related?* RQ1.3: RQ3: What is the correlation between the detection techniques based on bad smells? [S12] RQ8: What tools are used in TDM and what TDM activities are supported by these tools? [S26] RQ1.3: What research areas are emphasized in the literature that reports studies of TD (technical debt) in the context of ASD (Agile Sofware Development)? [S27]
Causality	Causality	C	9 *(22.5%)*	14 *(7.73%)*	*Does X cause Y?* RQ4: What evidence is there that Code Bad Smells indicate problems in code? [S11] RQ4: Which quality attributes are compromised when technical debt is incurred? [S26] RQ1: do all of the code smells equally impact software quality in terms of detected software defects? [S37]
	Causality-Comparative	CC	5 *(12.5%)*	6 *(3.31%)*	*Does X cause more Y than does Z?* RQ3.1: Attention level in the formal versus grey literature: How much attention has this topic received in the formal versus grey literature? [S16] RQ2: How similar/different are the experimental settings of studies investigating smell effect? [S24] RQ3.3: Considering the co-occurrence of bad smells in the papers of our dataset, how many of them actually study some relations between bad smells and what are the main findings of these co-studies? [S40]
	Causality-Comparative Interaction	CCI	0 *(0.0%)*	0 *(0.0%)*	*Does X or Z cause more Y under one condition but not others?*
Design	Design	D	0 *(0.0%)*	0 *(0.0%)*	*What's an effective way to achieve X?*
			Total	181 *(100.0%)*	

Table 11
Distribution of studies based on RQs focus.

Focus	Studies
Co-occurrence between smells and relationship with design patterns	S1, S28, S39, S40
Smell detection tools	S2, S4, S11, S13, S15, S18, S19, S20, S31, S32, S35, S39, S40
Model refactoring	S3, S29, S36
Refactoring techniques applied on smells	S4, S6, S8, S13, S17, S21, S22, S25
Trends, opportunities, challenges, gaps (refactoring and smells)	S5, S15, S17, S20, S21, S23, S24, S26, S27, S34, S37, S40
Refactoring tools	S6, S17, S25, S40
Software quality and refactoring (impact, attributes, measures, scenarios)	S7, S11, S15, S37
Product lines (refactoring, smells)	S8, S14
Clones	S9, S19, S20, S30, S31, S32, S33
Refactoring process (human knowledge, mental model)	S10, S22
Smells definition	S2, S4, S8, S11, S12, S15, S40
Smell detection approaches	S8, S11, S12, S13, S15, S18, S19, S20, S23, S26, S30, S32, S33, S34, S38, S39, S40
Introduction smells on systems, impact and affect/effect	S15, S24, S35, S37, S38, S39, S40
Tests	S16
Search-based	S17, S25
Technical debt	S21, S23, S26, S27

Revising the list of RQs can help researchers to use RQs for new secondary studies, creating better and more systematic RQs. We note that Description-and-Classification (DC) RQs are the most popular, with a wide margin (75% of studies). RQs of this type is mainly used in secondary studies. The second most frequent is the Frequency Distribution (FD), with Existence (E) and Descriptive Process (DP) in the third.

We identified 152 RQs (83.97%) about Exploratory and Base-rates Questions. In more detail, 34 studies (85%) have at least one RQ in these categories.[9] Likewise found in SLRs, these categories of RQs are more common in SMs and Surveys (Kitchenham et al., 2009; 2010b). Although, according to Kitchenham et al. (2010b), this distinction between SMs and SLRs is a bit confusing.

Relationship (R) and Causality (C) were a minority of the questions. It is the type of question one would ask to assess the effectiveness of treatments as in the traditional form of SLRs (Kitchenham et al., 2010b).

Our research evaluates the presence of empirical and evidence-based SE in secondary studies. We searched for terms like "empirical", "evidence", and "experimental" in all RQs, finding only 9 RQs (4.97%) related to this purpose. As we know from the growth of primary studies with such objectives, we suggest more secondary studies crossing and analyzing this information.

As shown in Table 10, to our surprise, **there were not RQs in any of the secondary studies of type Causality-Comparative Interaction (CCI) nor Design (D)**. One could identify such RQs as more sophisticated ones compared to the others: *e.g.,* a CCI RQ may look like this: *"Does smell A or B cause more maintainability problems under one condition but not others?"*. We hope to see secondary studies with such RQs in the future.

Also, We analyze the studies based on their RQs (and focus, when the study does not present RQs explicitly), and focus given on the defined RQs. We present the studies grouped by focus (Table 11).

Smell detection techniques are explored by several studies [S8, S12, S12, S13, S15, S18, S19, S20, S23, S26, S30, S32, S33, S34, S38, S39, S40]. There is a particular interest in smell detection approaches, using the most different approaches (we discussed the most previously mentioned). In some ways, some studies address techniques in specific contexts (*e.g.,* studies [S20, S30, S32, S33] in the context of clones).

Some studies have discussed techniques but not always ways to automate them [S8, S12, S23, S26, S30, S33, S34, S38]. Other studies have explored ways to automate detection techniques [S2, S4, S11, S13, S15, S18, S19, S20, S31, S32, S35, S39, S40].

Other studies have discussed trends and challenges in smells and refactoring topics [S5, S15, S17, S20, S21, S23, S24, S26, S27, S34, S37, S40]. Here, we do not differentiate them because we try to analyze both topics together, evaluating the relationship between them.

There are also studies related to specific topics not commonly mentioned (such as tests [S16] and model refactoring [S3, S29, S36]).

There is no relation between study comprehensiveness and the number of RQs: the scope of the study is related to the scope of the RQ itself and not necessarily with the number of RQs. Such relation does not appear in studies that have a large number of RQs ([S20] has 12 RQs, [S25] has 12 RQs, and [S17] has 10 RQs), but study [S15] reinforces this finding. It has 5 RQs but it is related to 6 focuses.

4.4.2. Ranking of cited secondary studies

To identify the highest-cited papers is becoming a popular subject not only in software engineering but in all computer science (Wohlin, 2007; 2008; Garousi and Fernandes, 2016a; 2016b). The reputation of the authors of a given paper could be a factor in our analysis. However, quantifying reputation is not easy, and discussing factors impacting the number of citations of a paper is outside the scope of our current work. To investigate the number of citations we used two classifications.

First, we adopted a citation metric: Absolute (total) number of citations since its publication. To obtain such citation data, we use Google Scholar mechanism: for each paper selected, we use the Google Scholar Search and save the number of citations. We show the top-five list of secondary studies based on the metric defined in Table 12.

Second, we adopt another citation metric: the number of citations per year, taking into account the year the work was published until now (Table 13). Comparing results considering the two metrics, we observed that the studies [S10, S20, S26] appear in both results. Also, among the topics most covered in the cited works are Code Clones [S20, S30] and technical debt [S23, S26]. Clones[10] have deserved more and more prominence from the community, with studies directed at the topic, and often being recognized as a specific research area. The TD[11] has also grown in recent years with

[9] See the data in the replication package (Lacerda, 2019).

[10] The term code clone generated about 545,000 results (about 17,200 in the last five years) in Google Scholar.

[11] The term technical debt generated about 2,490,000 (about 107,000 in the last five years) results in Google Scholar.

Table 12
Top 5 selected studies sorted by the number of citations of Google Scholar.

#	Title	Year	Citations
S10	A survey of software refactoring	2004	1301
S30	Survey of Research on Software Clones	2007	277
S20	Software clone detection: A systematic review	2013	216
S26	A systematic mapping study on technical debt and its management	2015	208
S11	Code Bad Smells: a review of current knowledge	2011	113

Table 13
Top 5 of selected studies sorted by the number of citations by year.

#	Title	Year	Citations by Year
S10	A survey of software refactoring	2004	86.72
S26	A systematic mapping study on technical debt and its management	2015	52.00
S14	A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring	2017	42.50
S20	Software clone detection: A systematic review	2013	36.00
S23	Identification and management of technical debt: A systematic mapping study	2016	27.00

studies focusing on management policies, tools, and techniques to mitigate its impact on software maintenance and evolution.

Forty studies had a total of 2568 citations, with an average of 64.2 citations per study and median=9. The difference between average and median gave by the fact that [S10] work has 1301 citations while [S30] has 272 ([S10] is almost 4.79 times more cited than [S30]). Since [S10] published in 2004, the number of citations was significant. Recent studies (published between 2017 and 2018) still has a small number of citations. Thus, it seems that primary studies more cited than compared to Surveys, SLR, and SM studies (Note that only 2% of cited studies refer to SLR/SM, as previously shown in Fig. 17, where they classified as "others").

Among the selected studies, [S10] remains the most cited with 14 citations. Next, the studies [S22, S3] appear with 13 and 12 quotes, respectively. Finishing the top 5 of the citations, we have the studies [S20, S11] with 11 and 10 citations. However, it is interesting to note [S3] is the most cited study among selected secondary studies (38.70% of the [S3] citations are from secondary studies).

RQ4 Summary

Challenges: Although the secondary studies aim at giving a panoramic view of the area, we identified the need for studies that contain more sophisticated RQs. With the increase of empirical studies on smells and refactoring, we consider new possibilities of secondary studies covering RQs about Causality (mainly CCI) and Design (D), comparing and evaluating phenomena, describing situations of efficacy and efficiency about methods, practices, and tools are open.

Observations: We had 181 RQs in the 40 selected secondary studies. The study [S40] with the highest number (13 RQs) is indeed the most comprehensive. Besides, we find studies with only one RQ. And, finally, studies that did not explicitly have RQs. It is the case of Surveys, which do not follow a defined protocol. Still, we had SLRs that did not follow a protocol too.

The scope of the study is related to the scope of the RQ itself and not necessarily with the number of RQs.

Code clones and technical debt are the recurring themes in the most cited secondary studies.

The study [S10] is the most cited among the secondary studies.

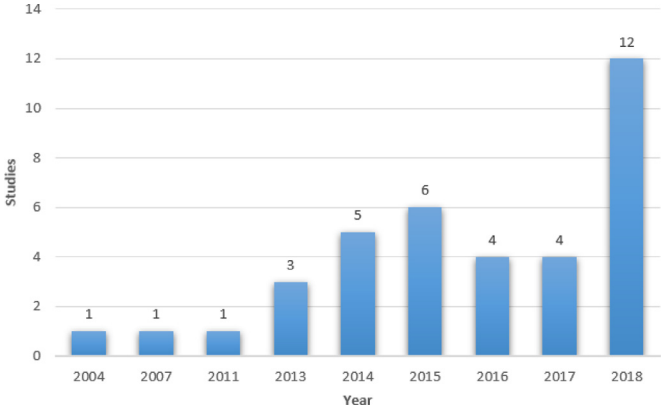

Fig. 15. Distribution of publications per year.

4.5. RQ5: What are the annual trends of types, quality, and the number of primary studies reviewed by the secondary studies?

The growing number of secondary studies on smells and refactoring is a strong indication of the high interest in this field. Next, we present some main characteristics of these secondary studies.

4.5.1. Paper types and references

Although our research defined as start reference the year 1992, the first secondary study [S10] published in 2004 (Fig. 15). Then we had just two secondary studies publications until 2012 (2007 and 2011). After 2013, the number of publications of secondary studies increased. We note it is due to the popularization of SLR and SM in the SE field. Of these selected studies, 75% published in journals and 25% published in conferences.

Most of the studies refer to SLR (65%) [S2, S3, S4, S5, S6, S7, S8, S11, S12, S13, S18, S19, S20, S21, S22, S24, S25, S27, S28, S29, S32, S34, S35, S36, S37, S40], with 17.5% refer to SM [S1, S14, S23, S26, S33, S38, S39] and 17.5% refer to Survey [S9, S10, S15, S16, S17, S30, S31]. Only one study is multivocal literature [S16] (Fig. 16). A multivocal literature mapping (MLM) (Garousi and Mäntylä, 2016) is an SLR that include data from multiple types of sources, *e.g.,* scientific literature and practitioners grey literature (*e.g.,* blog posts, white papers, and presentation videos). Multivocal mapping studies have just recently started to appear in SE literature.

In the snowballing process, we revised 7141 references, finding 184 studies. Among them, we selected seven studies to complement our research, using our selection criteria. Most of the studies (65%) did not use snowballing [S3, S4, S5, S6, S7, S8, S9, S10,

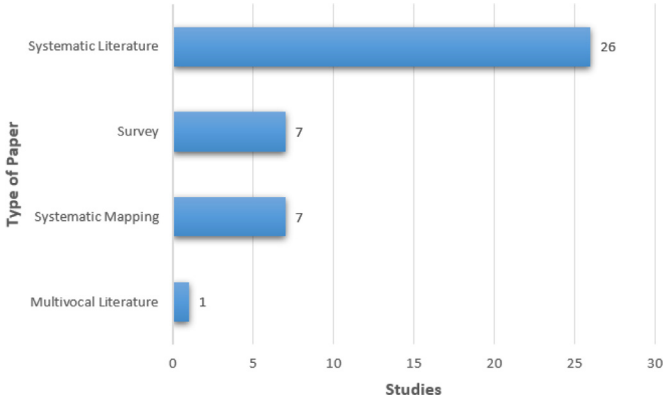

Fig. 16. Type of secondary studies.

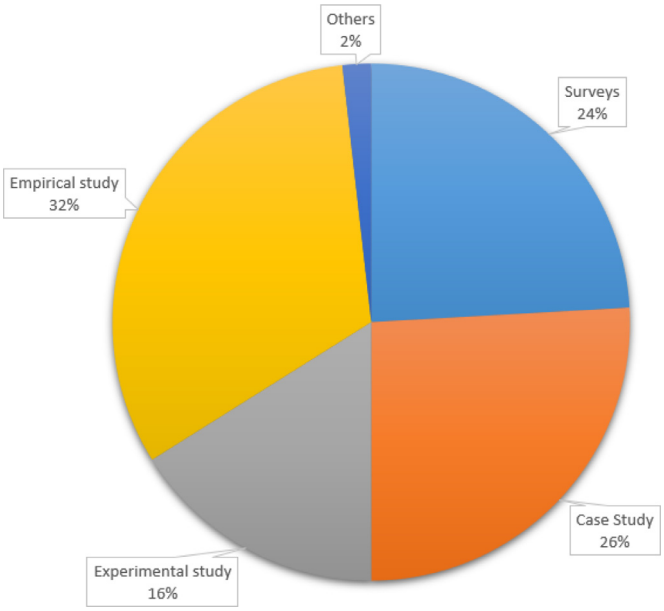

Fig. 17. Distribution of studies types cited by secondary studies.

S11, S12, S13, S14, S15, S19, S20, S22, S24, S28, S29, S30, S31, S32, S33, S36, S37, S38] and 35% used snowballing [S1, S2, S16, S17, S18, S21, S23, S25, S26, S27, S34, S35, S39, S40] as a research mechanism (Wohlin, 2014). We noted the **snowballing process appeared in studies published after 2015**. If we consider 2015 as a starting point, the use of snowballing is present in 50% of the selected studies, showing the growth of the snowballing process in secondary studies.

Most of secondary studies used (around 58%) empirical and case studies (Fig. 17). These types of studies are the preferred approaches for validating tools/prototypes. However, **there is a lack of validation using experts in this field**, qualifying the analysis. To increase confidence in empirical evaluation results in primary studies, we compile some information about secondary studies [S7, S22, S40]. According to such studies, it is necessary to pay attention to the following points:

1. Use a relatively large dataset implemented with different programming languages and considering a mixture of open-source and industrial systems;
2. Clearly define the study goal (smell detections, refactoring techniques) considered;
3. Fully identify the evaluation measures;
4. Adequately describe the study participants;

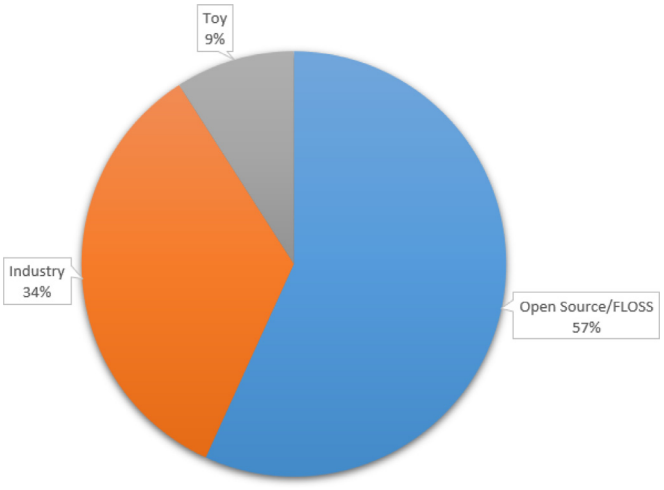

Fig. 18. Type of referenced projects.

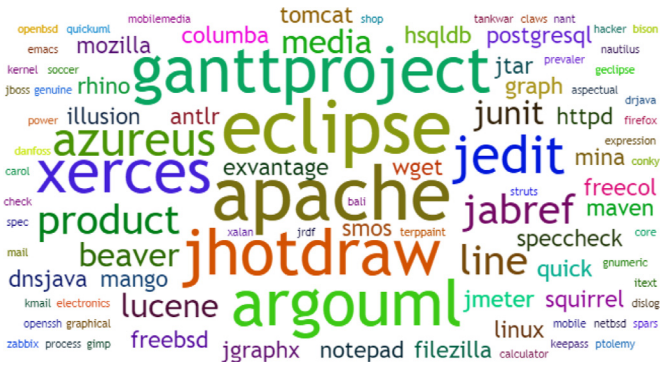

Fig. 19. Word cloud of projects cited in secondary studies.

5. Clearly describe the scoring systems adopted;
6. Compare the results with previous findings; and
7. Discusses validity threats.

According to the studies [S5, S6, S13, S22, S39, S40], although most researchers in this area are from academia, some participants in reported empirical studies are practitioners from industry, indicating they have some contact with each other. We found three studies [S17, S39, S40], where the authors from the academic and industrial sectors worked together. In Section 6, we discuss in more detail the implications of researchers and practitioners on this type of work.

4.5.2. Projects

Most of the academic researchers use nonindustrial datasets for the studies [S22, S39, S40], usually open-source/FLOSS and Toy applications as illustrated by (Fig. 18). The **majority of projects realized experiments with FLOSS**, representing 57% of the projects mentioned in the studies.

A tags cloud containing keywords of selected studies provides a high-level picture of the cited software projects (Fig. 19). The projects more cited in studies are *Apache, Eclipse, jHotDraw, ArgoUML*, and *GanttProject*. Most of the more quoted projects have some common features like a) they are long-lived projects (10+ years); b) they are structured in sub-projects; c) they are large projects, and d) they considered as marks in the open-source/FLOSS world.

According to [S18, S22], many authors perform experiments on open source projects for the evaluation of their techniques. Experiments on commercial/industrial projects are performed only by

few authors. On the one hand, it is easier to conduct experiments using FLOSS projects due to the availability of versions and constant evolution. More, many projects have often used as a basis for experiments. On the other hand, it is essential to put an emphasis on experiments carried out with industrial projects. It is a challenge to be faced in the coming years. It may also be essential to analyze the ratio of code smells existing in open source versus industrial projects. **We do not have until now a benchmark of projects** [S22, S39, S40]. Some studies [S13, S18, S39, S40] point a lack of benchmark definitions for smells validated by experts. It happens with refactoring, too. The large set of tools and systems used in the experimental settings suggest the lack of well-designed benchmarks should be better addressed. The benchmarks could be constructed, having the same characteristics as the most used systems.

PROMISES[12] is an example of a benchmark and an excellent initiative. However, such a benchmark does not seem to be updated for some time and also does not have specific datasets for refactoring and smells. Similar initiatives should be encouraged to contribute to advances on this topic.

4.5.3. Analysis of selected studies

The selected studies have as the initial year of research 1990 (through 2010) and as the final year varying from 2004 to 2018. The average period used to search the selected secondary studies is 15.2 years, with an average of 1444 papers considered and 92.1 papers selected by the studies. The study [S8] considered the minor time was seven years (2007–2014) of a total of 165 studies, picking 18. The study [S40] considered the most extensive-time period of research (27 years, from 1990 until 2017), considering 9633 papers, selecting 351 for analysis and discussion. The study [S34] has regarded as the most significant number of primary studies (13769), selecting 78. The study [S15] selected the most significant number of primary studies (445), of a total of 1028. The study [S16] had the highest accuracy in the survey (52.86), selecting 166 studies out of a total of 314. The study [S34] had the lowest efficiency (0.56), choosing 78 studies out of a total of 13769. Probably, in all cases, this is a consequence of the selection criteria used by the authors. The RQs used in the research has driven the study focus. *A priori*, studies more comprehensive have more RQs, but in fact, it depends on the RQ comprehensiveness. The comprehensiveness of a study is directly related to the statement of the RQs.

We also analyze if studies have search strings and inclusion/exclusion criteria explicit. Studies adopted different ways to explicit their search strings. Almost all studies without RQs also do not have explicit search strings and inclusion/exclusion criteria. Still, some studies have defined inclusion/exclusion criteria but did not have defined an explicit search string. It shows that some authors did not explain the protocol of the study, making it difficult for its reproduction. A better decision would be to adopt a research protocol, such as those recommended by Kitchenham et al. (2007, 2009).

The number of primary studies referenced can vary. A large number of regular surveys did not explicitly report the number of primary studies. In these cases, we counted the number of references of secondary papers, and use it as an estimate of the size of the study. The secondary studies have a sum of 4573 references, with an average of 114.32 references cited per study (median=99). Such data illustrated by Fig. 20, which presents the number of primary studies analyzed in each secondary research, grouped by publication year.

As previously discussed (Sub-Section 3.4), each Survey, SM, and SLR was evaluated using a set of quality-related criteria used in earlier studies. For each study in our pool, the quality score calculated by assigning {0, 0.5, 1} to each of the four questions. The result value in the range of (0, 4) where 4 is the maximum score.

[12] Research dataset repository is specializing in software engineering. More info: http://promise.site.uottawa.ca/SERepository/.

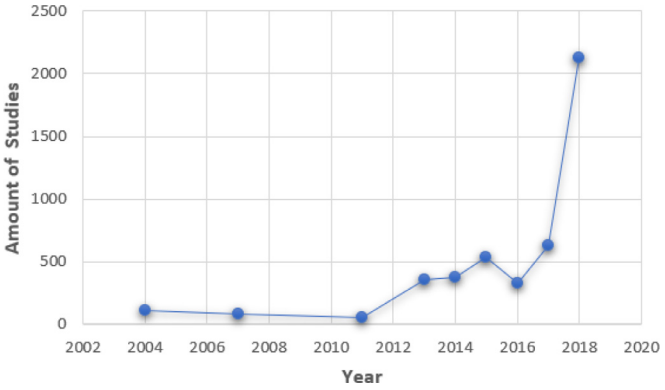

Fig. 20. Number of primary studies referenced by secondary researches per year.

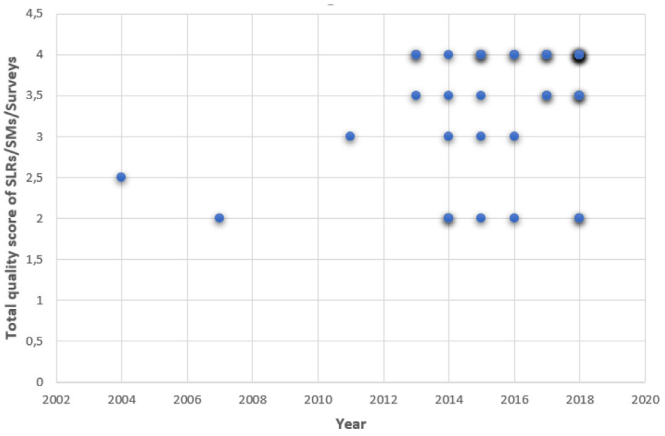

Fig. 21. Total quality score per year.

As shown in Fig. 21, all selected studies scored between 2 and 4. Surveys scored lower because they do not follow a formal protocol like SLR and SM do (even though some studies do not have RQs or selection criteria defined). Still, they are very cited in the literature. **More than 80% of the studies we selected scored between 3 and 4**.

RQ5 Summary

Challenges: As we previously noticed on tools analysis, there is a lack of validation by experts in the field: experts should participate more effectively in experiments.

Another challenge is the lack of benchmarks. It could enrich research and experiments, as well as support improved recall and accuracy of techniques and tools for both smells and refactoring.

Observations: We noticed a growth of secondary studies in recent years. Considering the last three years, we have more than half of secondary studies than previously published.

The vast majority of studies are SLR (65%) – 75% of the studies published in journals. The snowballing is not yet as present as the inclusion mechanism of new studies in the pool (it considering 2015 to 2018, 50% of studies did not use the process).

Fig. 22. Code smells and refactorings with some of their similar features. In the center, the aspects that connect them, by highlighting the quality.

The preferred approaches for validating the proposed tools/prototypes are empirical studies and case studies.

The selected studies point to FLOSS projects (*i.e., Apache, Eclipse, jHotDraw, ArgoUML*, and *GanttProject*) as the most used for experiments.

We do not have a recognized benchmark of projects until now.

5. The relationship between code smells and refactoring

As we have seen in the studies reviewed, code smells have certain interesting features. Code smells are symptoms or design problems that may affect the evolution and maintenance of the software. Some of these code smells are small, and we call a simple smell. Often, the occurrence of one code smell may be related to or correlated (as shown in Fig. 10) with another code smell, deriving a composite smell, or design smell. We have discussed in previous sections (see Sections 4.2 and 4.3.2) the main approaches for the detection of code smells, as well as the most cited tools to support this activity.

We also summarized some interesting characteristics of refactoring (see Sections 4.1 and 4.3.3). We note that there are simple refactoring, known as primitive refactorings. Often, for a given situation, we need to perform a sequence of refactoring, known as composite refactoring. Also, we present the more known tools supporting refactoring.

As we showed earlier, looking at code smells and refactoring in isolation, we noticed that there are some similar characteristics.

Code smells and refactoring affect software quality (see Sections 4.1.3 and 4.2.4). Quality is one of the most critical issues in software engineering, drawing attention from both practitioners and researchers. Developing software with quality is essential, but **preserving or increasing software quality during maintenance is even more critical**. Code smells produce software quality problems. Firstly, external quality attributes suffer from it over a long time, affecting the evolution of software, leading to increased technical debt. Second, internal quality attributes are also affected by code smells. Some code smells produce problems like low cohesion, high coupling, encapsulation-related problems that influence design decisions and maintenance. Refactoring and code smells are linked, because refactoring are the main strategy to remove/mitigate code smells, improving the software quality (clarity, simplicity, comprehension). We know, as discussed in the Sections 4.1.4 and 4.2.4, that refactoring is the primary approach to mitigating technical debt. We also know that refactoring, if improperly applied, can generate new code smells and, consequently, affect the quality negatively.

In Fig. 22, we present a scheme of this relationship, with the specific characteristics of code smells and refactoring, as well as the elements that connect them. So, code smells and refactoring are closely related to software quality.

5.1. Quality models, code smells and refactoring

Different software quality models are found in the literature and referenced in the studies [S21, S23, S26, S27, S39]. Each model defines a set of main software quality attributes. Some attributes are common to different models. **The models mentioned in the studies were ISO/IEC 9126** (International Standards Organisation (ISO), 1991), **FURPS**,[13] and **McCalls Factor Model** (McCall, 1977). However, the primary model of software quality factors mentioned in the selected studies is the ISO/IEC 9126. The ISO/IEC 9126 model is the most comprehensive, providing six main features classified as external attributes (*e.g.,* Functionality, Reliability, Usability, Efficiency, Maintainability, and Portability). Meanwhile, the current industry standard, called ISO/IEC 25010, mentioned in only one study [S26].

Another difficult task investigated in the software refactoring field is the preference of code smells to be corrected, based on given importance [S25]. A few studies [S15, S24, S35, S37, S38, S39, S40] identify the relationship between the detected types of smells and quality attributes. The nature of the relationship identified by the authors varies from one to another. **Most of the code smells**, in particular, defined by Fowler et al. (1999) **affect more than one quality attribute**. Therefore, **some quality attributes influence more than others**. The quality attributes most affected are maintainability, complexity, and understandability. They have a significant role in software maintenance costs. In this case, the **set of code smells related to these quality attributes will have the highest degree of priority for removal from the software**.

On the other hand, only one study [S7] has explored the impact of refactoring on quality attributes. The study [S7] presented external and internal quality attributes. The external quality attributes more often are maintainability, reusability, and understandability. Reliability and maintainability are attributes more studied. The internal quality attributes more investigated are cohesion, coupling, complexity, inheritance, and size attributes, where coupling and size are the most and least considered attributes, respectively. Coupling measures have also been one of the main approaches to evaluate decay [S38] and technical debt [S21]. It is important to note that estimated external quality attributes are quantified using combinations of internal quality measures such as cohesion, coupling, and inheritance. Therefore, studying **the impact of refactoring on a single internal quality attribute is potentially more straightforward and more accessible than addressing combinations of internal quality attributes**. The study [S7] also recommends that

[13] https://en.wikipedia.org/wiki/FURPS.

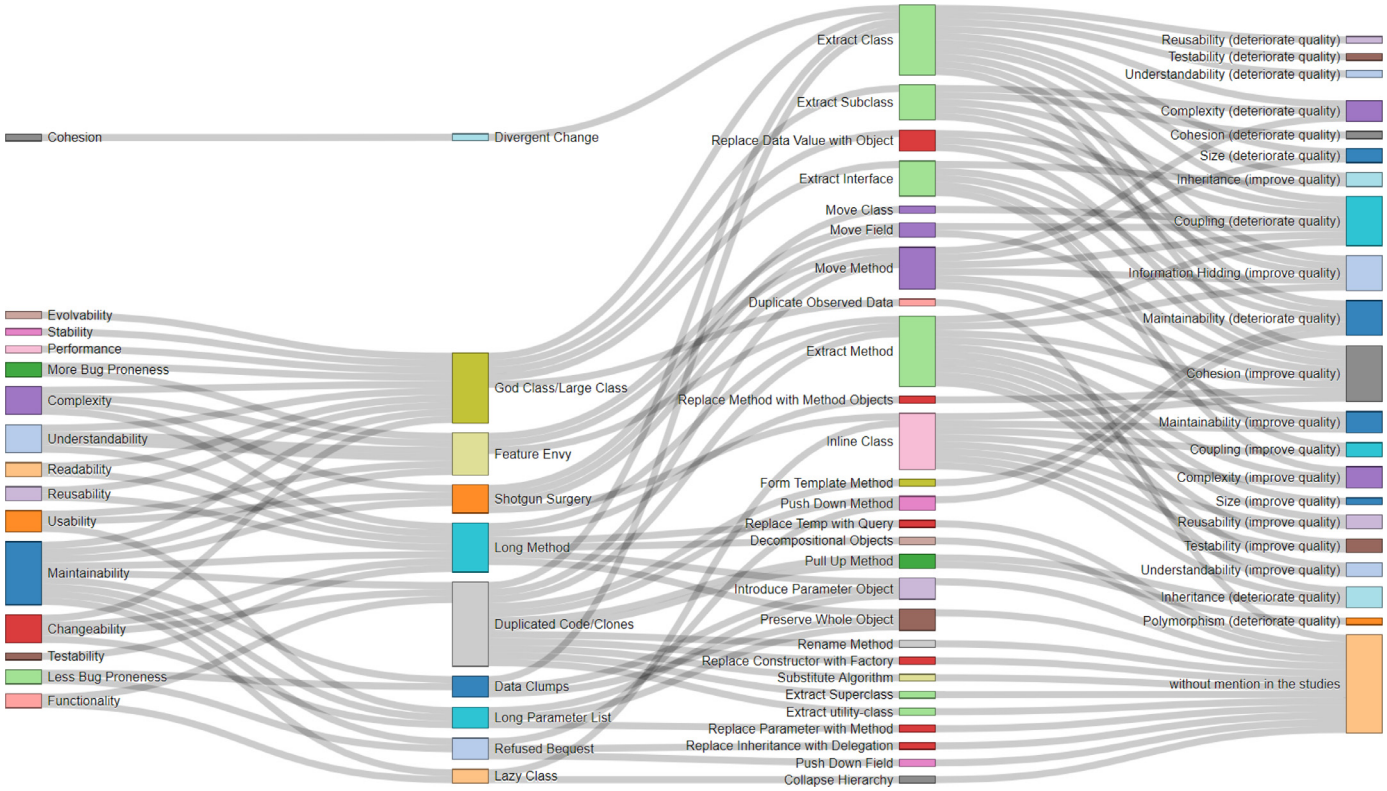

Fig. 23. The relationship among (external and internal) quality attributes and their impact on code smells and refactoring.

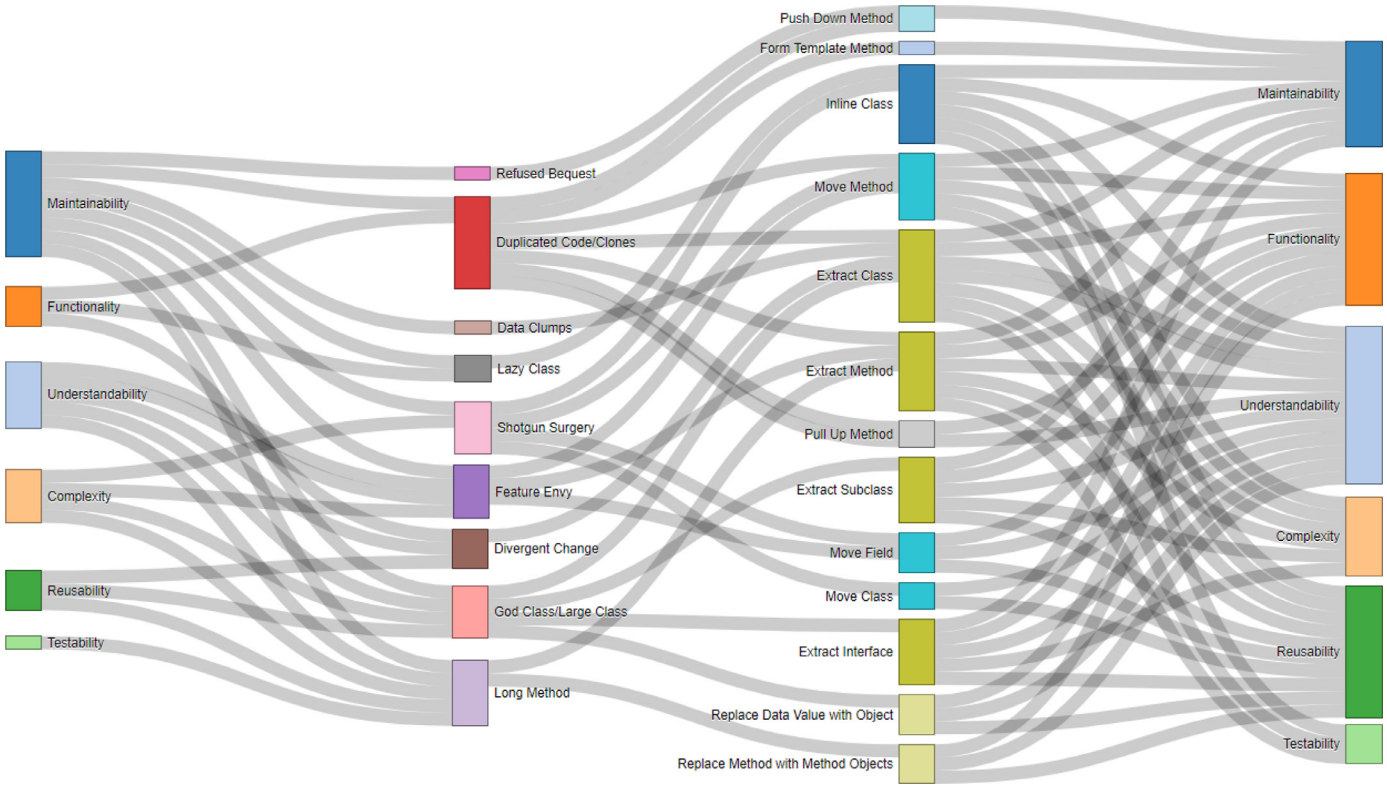

Fig. 24. Relationship among quality attributes using the QMOOD model, impact on code smells and refactoring.

researchers conduct more empirical studies to explore the impact of refactoring on external quality attributes because these attributes are of direct interest to practitioners.

According to studies [S7, S22], the researchers observed that **different refactorings potentially have different, and sometimes conflicting, impacts on quality**. It is difficult to distinguish between the effects of individual refactoring scenarios or to draw any conclusions regarding their impacts on quality. **One recommendation is must apply a set of refactoring that follow the same scenario and assess quality before and after refactoring**.

5.2. Analysis

We found studies [S15, S24, S35, S37, S38, S39, S40] focused on verifying how the occurrence of code smells impacts several quality attributes. In the same way, we found one study [S7] discussing refactoring and their impact on quality. With a base on results, we develop a visualization (Fig. 23), that allows analyzing the relationship among quality attributes, which code smells affected them, which refactoring can be applying to these code smells, and what is the impact on quality when using such refactoring.

The **attributes of quality most affected by code smells are maintainability, understandability, and complexity**. However, evolvability, stability, performance, and testability mentioned in only one code smell, each one.

The code smells that most affect different quality attributes are *God Class/ Large Class, Long Method* , and *Feature Envy* . *God Class/Large Class* is the most affect quality attributes. It affects maintainability, complexity, evolvability, stability, performance, readability, reusability, and changeability. *God Class/Large Class* and *Feature Envy* are more prone to bugs, as well as affecting complexity, understandability, usability, and maintainability. *Long Method* affects complexity, understandability, readability, reusability, maintainability, changeability, and testability.

Some code smells have a larger set of refactoring than others (see Table 8). It is the case of ***Duplicated Code/Clones*, which have more refactoring that can be applied**. In this smell, we can use *Pull Up Method, Rename Method, Replace Constructor with Factory, Form Template Method, Pull Up Method, Push Down Method, Substitute Algorithm, Extract Superclass, Extract Class, Extract utility-class*, and *Move Method. Long Method* presents an alternative the refactoring *Extract Method, Replace Temp with Query, Introduce Parameter Object, Preserve the Whole Object, Replace Method with Method Objects*, and *Decompositional Objects. God Class/Large Class* also presents some refactoring, such as *Extract Class, Extract Subclass, Replace Data Value with Object, Extract Interface*, and *Duplicate Observed Data*.

According to previously presented, the relationship between refactoring and code smell is not one-to-one [S7]. **Refactoring are flexible, can be applied in more than one code smell**. It is the case of *Extract Class, Move Method*, and *Extract Method*, which also have been more studied by researchers.

We also commented that refactoring does not always improve all quality attributes. **When quality improvement is the goal of refactoring, developers should be careful and check whether the application's proposed refactoring achieves the desired goal**. Developers need to know they apply such refactoring on smell, which can lead to the introduction of other ones. It is important to note that by positively affected by refactoring on a quality attribute, we mean that the refactoring causes the value of measure that quantifies the quality attribute to increase, and vice versa. However, increase the value does not always mean that the quality is improved because, for some quality attributes (*e.g.,* coupling, complexity, size), the improvement indicated by the decrease in the corresponding value. Of course, it depends on how being calculates such a metric.

Refactoring also affect different quality attributes. For instance, ***Extract Method*and *Extract Class*are refactoring that most affect different quality attributes** (10), with *Inline Class* affected 8 attributes. *Extract Method* affects inheritance and coupling negatively. The same refactoring affects complexity, cohesion, size, information hiding, maintainability, reusability, testability, and understandability positively. *Extract Class* affects inheritance, cohesion positively, and information hiding. Also, it affects coupling, complexity, size, maintainability, reusability, testability, and understandability negatively.

It is also important to mention that **14 refactoring options for the presented code smells do not have studies associated with their impact on quality**. It is the case of *Replace Constructor with Factory, Substitute Algorithm, Extract Superclass, Extract utility-class, Introduce Parameter Object, Duplicate Observed Data, Replace Temp with Query, Preserve Whole Object, Decompositional Objects, Replace Parameter with Method, Replace Inheritance with Delegation, Push Down Field, Collapse Hierarchy*, and *Rename Method*. Therefore, we do not know their impacts when applying these refactoring.

Besides, when evaluating the quality, **developers are advised to consider several quality attributes and not focus on one particular attribute**, ignoring others. Otherwise, the proposed refactoring may detract from quality rather than improve it. We show that the impact of refactoring on most of the measured external quality attributes has not studied. Consequently, the impact of refactoring on measured external quality attributes requires more research and study. We recommend that researchers conduct more studies to explore the impact of refactoring on external quality attributes because these attributes are of direct interest to practitioners.

The studies [S7, S24, S35, S37] are recent and **drive the necessity to answer if code smells harms the project, as well as the impact on refactoring**. Moreover, the **number of quality attributes and their possible combinations with distinct code smells are high, thus requiring different studies. The same happens with refactoring**. The study of impact/effect has been receiving so much attention until now. It suggests that there is still no comprehensive and sufficient evidence on the extent of adverse effects associated with code smells and positive impact on refactoring on software maintenance and evolution.

In studies on code smells, more external quality attributes referenced. In refactoring studies, there were internal and external attributes. Of course, about refactoring, we may experience code deterioration or improvement, as we discussed earlier. To better represent this relationship between internal and external quality attributes, we have grouped internal attributes with their proper relation to external attributes. For this, we use the QMOOD model (Bansiya and Davis, 2002). We consider only the internal attributes found in the secondary studies and, based on QMOOD model, observe where they applied. We also ignored the quality attributes referenced by QMOOD model but not mentioned in the studies used in our research. Thus, we define that coupling used in reusability and understandability. We observe that both cohesion and size used in reusability, understandability, and functionality. Understandability uses information hiding. Functionality and understandability use Polymorphism. As shown in Fig. 24, we present a new view with a different arrangement of quality attributes.

We note that the quality attributes most affected by code smells are also the ones most affected by refactoring. We identify that refactoring have more impacts on understandability, functionality, reusability, and maintainability. With this new redistribution, we note that by grouping internal attributes with external attributes using the QMOOD model, **refactoring affect quality more than code smells**. It demonstrates that **this relationship is not only due to the code smell refactoring link, but that the origin of the relationship is quality**. And that in the medium to long term, depending on the context of refactoring, can lead to new

code smells, generating a cycle that can further deteriorate the software. Therefore, we recommend that other studies explore this relationship of code smells and refactoring quality as their main aspect.

6. Implications

We now discuss the implications of this systematic literature review on code smells and refactoring for practitioners, researchers, and instructors because, as explained by Goues et al. (2018), it is essential that SLRs provide some advice beyond their RQs.

Practitioners Real software systems must be continually changed to meet the demands of the market and the expectations of their users. Thus, software development teams must be always concerned by the continuous improvement and quality of their systems (Canfora et al., 2011). However, there are still many challenges related to software maintenance and evolution, including the need to understand the systems and the complexity involved in the development process. The lack of understanding of architectural deviations during software evolution compromises both the development process and the systems themselves.

Leppänen et al. (2015a) present a decision-making framework, based on practitioners' perceptions. They claim that the need for refactoring is rather subjective and not necessarily rational. They show the empirical nature of the refactoring process. Exposure to the real world brings invaluable insights (Griswold and Opdyke, 2015; Leppänen et al., 2015b; Lahtinen and Leppänen, 2016). Developers are the real specialists who, with their perceptions and experiences, can say which structure is better than the others. Their knowledge should drive refactoring. We argue that refactoring should be a daily habit. The more refactoring are neglected, the greater the likelihood and need of doing larger refactoring, which are more problematic: they must be planned, they interrupt daily work, and often they must be justified to management. We reported in this paper that the most applied refactoring are primitive refactoring. Indeed, they are simpler than composite refactoring and are automated in tools.

The use of version control, testing, and reviews are encouraged as good practices (Leppänen et al., 2015b). Code reviews, for example, can help find targets for refactoring. We observed[14] that these practices help reduce code complexity, maintain quality, and improve the source code in the long run. New research discussing these practices in relation to refactoring could bring new perspectives on how developers should address such good practices.

Developers know the values of refactoring but are often prevented from applying them (Tempero et al., 2017). One of the reason preventing the use of refactoring is the lack of measures showing their impacts. The lack of monitoring of refactoring was also pointed in several works (Mens et al., 2003; Leppänen et al., 2015a; 2015b). We observed that, when prioritizing code smells, seeing the relationship and the impact of quality attributes to code smells helped their prioritization. When performing refactoring, it is also necessary to assess their impacts.

Instructors: The implications described for practitioners are also valid for instructors.

The concept of "quality" is present in all software engineering knowledge areas (KAs) (Society et al., 2014). We highlight that design, construction, testing, maintenance, models and methods, quality, and computing foundation KAs associating topics discussed in this work. We present a pragmatic way to support instructors in their classes.

The discussion about code smells and refactoring began with the introduction of eXtreme Programming in curricula (Goldman et al., 2004). Some studies (Smith et al., 2006; Stoecklin et al., 2007) described experiences in applying some lessons learnt based on self-documenting and functional tests, encapsulation, and unit testing, refactoring of constants and variables and to extract methods. These studies did not include all the lessons needed to learn refactoring, but reported some benefits, like the importance of self-documenting code, code smell recognition, testing comprehension, and improvement of code style. Then, several approaches were proposed to support instructors in learning/teaching refactoring, including tutoring systems (Haendler et al., 2019), agents (Sandalski et al., 2011), gamification (Haendler and Neumann, 2019), and on-line teaching (López et al., 2014). Yet, it is necessary to use real-world examples for learning and teaching. The use of a existing source code with real (or injected) problems promotes a collaborative environment to exchange knowledge.

We identified that these topics should be included in various courses in curricula, such as introduction to programming, software engineering, software quality, software design. We observed that the exposition to change brings students the perception of concepts that are progressive and intuitive. The reasoning behind refactoring is also essential (*when do I refactor? What do I refactor? Why is this refactoring better?*). When refactoring are taught and used in class, they help students to acquire good programming practices and design principles.

We reported several aspects that instructors could consider in their classes because we are instructors ourselves. We have had the opportunity to apply and to discuss code smells and refactoring in classes. We taught this topic in undergraduate[15] and graduate courses.[16] One of the practices that we use and recommend is to perform *Coding Dojos*.[17] A *Coding Dojo* allows students to learn practices, such as test-driven development, refactoring, code review, pair programming, and the use of tools such as static analysis, code coverage, and build tasks. We encourage instructors to incorporate both code smells and refactoring into their classes to develop students' technical skills. New developers should be aware of code smells and how to remove them.

Researchers We sought to show the close relationship between code smells and refactoring. We compiled the main results of the secondary studies and highlighted several challenges.

We understand that refactoring exist because code smells indicate that something is not right in the source code. Thus, a standardized form of code smell definitions, detection methods, and tools is needed. Dig (2017) showed that there is growing interests on automation, prioritization, inference, and recommendation of refactoring. Researchers are encouraged to explore such interests together with the practice of refactoring. They should focus their research on refactoring that are often applied in practice rather than study refactoring opportunities already considered in the literature and–or rarely applied in practice. Researchers should also explore improving the results of applying refactoring that are commonly used in practice as well as propose refactoring tools.

Also, researchers should analyze the impacts of code smells and refactoring on software quality and technical debt. They must strive to bring their research closer to the software industry. Recent studies (Garousi et al., 2019; 2019; Garousi et al., 2019) on industry and education in software engineering showed a large gap in several areas, including quality and design. These areas are closely related to the topics discussed in this study. Researchers

[14] One of the authors of this work worked for 20+ years in the industry, helping software development teams to improve code quality. This perception is based on his experiences.

[15] Object-Oriented Programming, Software Engineering, and Software Architecture at UniRitter; Techniques of Program Construction at UFRGS.

[16] Smells, Patterns and Refactoring at UniRitter; Agile Development Introduction and Agile Development with eXtreme Programming at Unisinos.

[17] http://codingdojo.org/.

Table 14
Open issues on smells, refactoring or both.

Issue	Topic	Observations
I1 - Code smell naming	Smell	There is an apparent problem of code smelling nomenclature [S11, S12, S15, S39, S40]
I2 - Approaches to code smell detection	Smell	We need to explore which approaches are most effective in smell detection [S2, S13, S15, S18, S19, S20, S30, S31, S32, S33]
I3 - Code smell and industry perception	Smell	We need to assess whether practitioners recognize code smells and whether practitioners know this code smells as symptoms of code problems
I4 - Developers and code smell detection process	Smell	It is essential to evaluate the participation of developers in the code smell detection process [S2, S11], using the developers' insights and experiences to improve the detection process
I5 - Code smell and impact on quality attributes	Smell	The studies [S15, S23, S24] do not establish an explicit connection between smells and quality, showing there is an opportunity for further studies
I6 - Code smell tools	Smell	There are many opportunities to explore smell detection tools [S2, S4, S9, S13, S15, S18, S19, S39, S40]
I7 - Developer's refactoring knowledge	Refactoring	The use of developers' knowledge about refactoring [S10, S22] can help to improve the refactoring process, refactoring tools, among others
I8 - Refactoring and impact on quality attributes	Refactoring	There are many opportunities to research a low explored refactoring, most commonly used refactoring by developers and their relationships with quality attributes [S4, S5, S7, S22]
I9 - Refactoring tools	Refactoring	There is an opportunity to propose/improve refactoring tools [S2, S9, S17, S18, S39, S40]
I10 - Refactoring tracking	Refactoring	There is a research opportunity on refactoring tracking and monitoring the effects of refactoring [S21, S23]
I11 - Architectural refactoring versus contextual refactoring	Refactoring	It is an open theme for researchers to explore where is this frontier about architectural and contextual refactoring (*e.g.,* package, class, method), as well as its benefits for quality [S21, S23, S26, S27]
I12 - Teaching code smell and refactoring	Both	Researchers can develop studies in the classroom about best practices and tools for learning and training
I13 - Research gaps on code smell and refactoring	Both	Production of reliable datasets, use of large-scale academic and industrial projects, industrial and academic analysis looking for data inconsistencies are examples of trends, opportunities, and gaps [S5, S15, S17, S20, S21, S23, S24, S26, S27, S34, S37, S40]

do work with industry: almost 50% of collaborations started in industry and 90% generated at least one paper (Garousi et al., 2019), which show the importance of industry-academic collaboration. Researchers should strive to create partnerships, showing how they can be useful for both areas.

7. Open issues

We now present some open issues, *i.e.,* questions for future works, concerning code smells detection, refactoring techniques, support tools, impact on quality, and academic research and its relationship with industry.

Mens et al. (2003) presented some future trends in research in 2003. After more than 15 years, some questions remain open, interesting topics for future works. We discuss in detail each identified issue, summarized in Table 14.

As we noted in relation to studies [S11, S12, S15, S39, S40], there is a problem of naming and organizing code smell (see Section 4.2.2). Code smells have definitions that are sometimes complex and not informal (Murphy-Hill and Black, 2008). Therefore, researchers could research the standardization of code smells.

As shown in Table 8, several studies [S2, S13, S15, S18, S19, S20, S30, S31, S32, S33] presented many approaches to code smell detection (see Section 4.2.3). However, we still must understand which approaches are the most effective. Some code smells have more than one approach while others have none. Researchers could study the combinations of approaches to code smell detection as well as propose approaches for currently undetected code smells.

There is an opportunity to assess whether the most cited code smells impact the industry. Recent studies (Taibi et al., 2017; Hozano et al., 2018) also evaluated the developers' perceptions and how they detect smells. Researches should also assess whether practitioners recognize code smells and whether practitioners know that code smells are symptoms of problems.

In the code-smell detection process, researchers should evaluate the developers' participation (*e.g.,* when, how) [S2, S11] because they are critical in this process (dos Santos Neto et al., 2015). Developers' insights and experiences can be explored in future work to improve the detection process.

Some studies [S24, S35, S37, S38] discussed the impact of code smells on quality attributes (see Section 4.2.4). Other studies [S15, S23, S24] did not establish an explicit connection between smells and quality, which is an opportunity for further work.

Following previous studies [S2, S4, S9, S13, S15, S18, S19, S39, S40], we organized a list of code smell detection tools (see Section 4.3.2). However, many of them are obsolete or have severe limitations (scalability, prioritization, visualization, multi-smells, multi-language, low precision, and recall). Researchers should invest into developing more comprehensive tools.

Research related to developers' knowledge about refactoring is essential [S10, S22] to understand the developers' mental models when refactoring. Although IDEs provide some automated refactoring, some studies showed that developers do not use them (Murphy-Hill and Black, 2007a; Vakilian et al., 2012) (see Section 4.3.3) because of usability or lack of knowledge of the refactoring process (Murphy-Hill, 2006). Researchers could evaluate whether the decision is rational or subjective, refactoring is a daily activity, factors used in the decision to refactor, learning, among others.

Some studies discussed refactoring and their impact on quality attributes [S4, S5, S7, S22] (see Section 4.1.3) but with small numbers of refactoring (Section 4.1.1) and different quality models (Section 5). Researchers could study systematically refactoring and their relationship to quality. Other studies could measure the impact of refactoring on individual attributes (*e.g.,* security, understandability, and extensibility) and studies to evaluate most commonly used refactoring by developers that affect the quality (decay or improvement), using quality models such as ISO 25010.

Some studies [S2, S9, S17, S18, S39, S40] presented refactoring tools. Table 8 and Section 4.3.3 show that there are few refactoring tools and some are obsolete. There is an opportunity to propose and improve refactoring tools, especially tools to predict and evaluate the effects of refactoring.

We showed in Section 4.1.3 that applying refactoring can cause problems [S21, S23]. Therefore, we identify a research opportunity on refactoring monitoring, to monitor the effects of refactoring on software evolution.

Some studies [S21, S23, S26, S27] discussed architectural refactoring and their benefits as well as whether refactoring should be contextualized (*e.g.,* method, class, and package). However, it is an open issues for researchers to explore the frontiers between contexts and their benefits for quality.

Teaching code smells and refactoring is interconnected (see Section 6). Researchers could develop studies in the classroom about best practices and tools that enhance learning, training future professionals with this awareness.

Code smells and refactoring are evolving research areas and some studies bring trends, opportunities, and gaps [S5, S15, S17, S20, S21, S23, S24, S26, S27, S34, S37, S40]. We suggest topics such as identifying reliable datasets that can be compared with existing studies, using large-scale academic and industrial projects to generate more reliable conclusions and analyzing industrial and academic studies looking for data inconsistencies.

8. Threats to validity

This section discusses some threats to the validity and some decisions to mitigate them. The search string of an SLR needs to very well defined to return secondary studies that are relevant to the search topic. In this study, we used several synonyms referring to the main terms of the SLR goal searched. Some pilot searches conducted to find new synonyms for the search string. Therefore, we believe the defined search string has returned as many relevant secondary studies as possible. Thus, we seek to broaden our research spectrum. However, not all topics covered by secondary studies. It also does not mean that the community does not include this topic.

The choice of electronic databases is another factor that may impact the results of an SLR. In this study, we perform a search for secondary studies in eight different electronic databases. Therefore, other databases not used in the survey may contain work that is relevant to this review. To reduce this threat, we do carry out a snowballing process to find more potentially relevant studies. In this step, the citations of the selected papers verified through a list of references to find pertinent other studies not included initially on our search.

This SLR considered only papers written in English. Some relevant studies may be written in other languages. However, the primary venues of scientific publication in SE accepts papers in English. Therefore, we consider that using English is sufficient to filter the main studies on the subject.

Besides, the classification scheme of studies is another point that considered a threat to validity. The data extraction was performed subjectively and grouped into categories to facilitate the reading and the understanding of the readers. To avoid bias, we follow some procedures. However, other reviews can have different classification schemas and ways to group and analyze the papers. Another threat is related to the granularity of the information presented in the reviewed secondary studies. If some information is not described in these studies, it may affect our conclusions. Reliability validity is concerned with issues that affect the ability to draw that the operations of a study can be repeated with the same results. Our research can easily replicated following the steps described and using the search string.

9. Conclusion and future work

We performed a tertiary study on code smells and refactoring. We systematically analyzed 40 secondary studies, answering five RQs to present the main challenges (*what we do not know*) and observations (*what we know*) related to code smells and refactoring. We summarize some of the main findings below.

We show that the majority of studies discuss smells and refactoring separately (Fig. 5). Only two secondary studies explore both explicitly. Smells have the majority of secondary studies (62.5%), with different focuses of study. *Duplicated Code/Clones* and *God Class/Large Class* are the most mentioned code smells. Also, *God Class/Large Class* has been the most investigated smells related to technical debt. We observe problems related to smells definition, affecting their detection.

Another challenge deserving more studies are the co-occurrence of code smells, that is, the appearance of code smell in consequence of another or code smells that are always close. There have several detection approaches, being metrics-based, and strategies/rules the most cited among them. Besides, smell detection approaches and the corresponding produced results are highly inconsistent. There is no consensus on the standard threshold values for the detection of smells, which are the cause of the disparity in the results of different approaches. Some approaches, like probabilistic/search-based, have grown and deserve attention.

Extraction refactoring (such as extract classes or methods) have been the most explored in the studies. Only a small set of refactoring have studied (around 27 of 72), opening up possibilities for studies that evaluate other refactoring, as well as assess the reason for not exploring them. Although refactoring has been an ally of developers to reduce technical debt, the refactoring do not always improve code quality.

We found 162 distinct smell detection tools, and *CCFinder* is the most cited one. Also, we found 24 distinct refactoring tools, and *JDeodorant* is the most cited one. We have also seen that there is a gap in the development of refactoring support tools, such as the extension of techniques not yet exploited (opportunities, execution, developer support, impact on quality). We cross-referenced the most frequently reported smells with detection approaches, detection tools, suggested refactoring, and refactoring tools (Table 8). We note that even though there are several smell detection tools, many are discontinued or have low accuracy. Similarly, we see that there is room to explore refactoring tools in these smells.

We present some open questions as a basis for future studies on smell detection, refactoring, supporting tools, impact on software quality, as well as research that considers both academia and industry (Section 7).

Our study shows quality attributes make relationships between refactoring and code smells. We noticed that the quality attributes affected by code smells were the same affected by refactoring. Code smells and refactoring have a relationship with understandability, maintainability, testability, complexity, functionality, and reusability. Besides, we also observed refactoring affect quality more than code smells.

The relationship between code smell and refactoring have several open questions. For instance, which refactoring could apply in a specific code smell? Which refactoring can be combined to mitigate such code smell? Which refactoring have the most significant impact on quality? Briefly, we suggest more studies to investigate refactoring and code smells jointly as a single phenomenon.

We hope this study could instigate researchers to investigate more deeply both practices and tools to mitigate the code smell and evaluate the impact on quality. In the same way, we suggest studies to explore the goal of refactoring used by practitioners and their effect on quality as well as the development/improvement of refactoring tools to monitor refactoring and its gains.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Guilherme Lacerda: Conceptualization, Methodology, Validation, Investigation, Resources, Writing - original draft, Writing - review & editing, Visualization, Supervision, Project administration. **Fabio Petrillo:** Conceptualization, Methodology, Validation, Investigation, Resources, Writing - review & editing, Supervision. **Marcelo Pimenta:** Conceptualization, Methodology, Validation, Writing - review & editing, Supervision. **Yann Gaël Guéhéneuc:** Validation, Writing - review & editing, Supervision.

References

Abbes, M., Khomh, F., Gueheneuc, Y., Antoniol, G., 2011. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: 2011 15th European Conference on Software Maintenance and Reengineering, pp. 181–190. doi:10.1109/CSMR.2011.24.

Abebe, M., Yoo, C.-j., 2014. Classification and summarization of software refactoring researches: a literature review approach. In: Advanced Science and Technology Letters. Science & Engineering Research Support Society, pp. 279–284. doi:10.14257/astl.2014.46.59.

Abebe, M., Yoo, C.-J., 2014. Trends, opportunities and challenges of software refactoring: a systematic literature review. Int. J. Softw. Eng. Appl. 8 (6), 299–318. doi:10.14257/ijseia.2014.8.6.24.

Aivosto, 2018. Project Analyzer v10.2 for Visual Basic, vb.net and vba. http://www.aivosto.com/project/

Al Dallal, J., 2012. Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics. Inf. Softw. Technol. 54 (10), 1125–1141. doi:10.1016/j.infsof.2012.04.004.

Al Dallal, J., 2015. Identifying refactoring opportunities in object-oriented code: a systematic literature review. Inf. Softw. Technol. 58, 231–249. doi:10.1016/j.infsof.2014.08.002.

Al Dallal, J., Abdin, A., 2018. Empirical evaluation of the impact of object-Oriented code refactoring on quality attributes: a systematic literature review. IEEE Trans. Softw. Eng. 44 (1), 44–69. doi:10.1109/TSE.2017.2658573.

Ali, A., Sulaiman, S., 2014. A systematic literature review of code clone prevention approaches. Int. J. Softw. Eng. 1 (JANUARY 2014), 1–6.

Alkharabsheh, K., Crespo, Y., Manso, E., Taboada, J.A., 2018. Software design smell detection: a systematic mapping study. Softw. Qual. J. doi:10.1007/s11219-018-9424-8.

Alves, N.S., Mendes, T.S., de Mendonça, M.G., Spínola, R.O., Shull, F., Seaman, C., 2016. Identification and management of technical debt: asystematic mapping study. Inf. Softw. Technol. 70, 100–121. doi:10.1016/j.infsof.2015.10.008.

Bandi, A., Williams, B.J., Allen, E.B., 2013. Empirical evidence of code decay: a systematic mapping study. In: Proceedings - Working Conference on Reverse Engineering, WCRE, pp. 341–350. doi:10.1109/WCRE.2013.6671309.

Bansiya, J., Davis, C.G., 2002. A hierarchical model for object-oriented design quality assessment. IEEE Trans. Softw. Eng. 28 (1), 4–17. doi:10.1109/32.979986.

Barrett, A., Edwards, J., 1995. Knowledge elicitation and knowledge representation in a large domain with multiple experts. Expert Syst. Appl. 8 (1), 169–176. doi:10.1016/0957-4174(94)E0007-H.

Beck, K., Andres, C., 2004. Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley.

Behutiye, W.N., Rodríguez, P., Oivo, M., Tosun, A., 2017. Analyzing the concept of technical debt in the context of agile software development: a systematic literature review. Inf. Softw. Technol. 82, 139–158. doi:10.1016/j.infsof.2016.10.004.

Besker, T., Martini, A., Bosch, J., 2018. Managing architectural technical debt: a unified model and systematic literature review. J. Syst. Softw. 135, 1–16. doi:10.1016/j.jss.2017.09.025.

Bian, Y., Su, X., Ma, P., 2014. Identifying Accurate Refactoring Opportunities Using Metrics. Advances in Intelligent Systems and Computing, 250. Springer India doi:10.1007/978-81-322-1695-7.

Brown, W.H., Malveau, R.C., III, H.W.M., Mowbray, T.J., 1998. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley and Sons, Inc, Canada.

Bruch, M., Monperrus, M., Mezini, M., 2009. Learning from examples to improve code completion systems. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, pp. 213–222. doi:10.1145/1595696.1595728.

Cairo, A.S., Carneiro, G.D.F., Monteiro, M.P., 2018. The impact of code smells on software bugs: a systematic literature review. J. Inf. Sci. Technol.Eng. 9 (October), 1–21. doi:10.20944/preprints201810.0059.v1.

Canfora, G., Penta, M.D., Cerulo, L., 2011. Achievements and challenges in software reverse engineering. Commun. ACM 54 (4), 142–151.

Cardoso, B., Figueiredo, E., 2015. Co-occurrence of design patterns and bad smells in software systems: An exploratory study. In: Proceedings of the Annual Conference on Brazilian Symposium on Information Systems: Information Systems: A Computer Socio-Technical Perspective - Volume 1. Brazilian Computer Society, Porto Alegre, Brazil, Brazil. 46:347–46:354

Chatzigeorgiou, A., Charalampidou, S., Ampatzoglou, A., Chatzigeorgiou, A., 2017. Identifying extract method refactoring opportunities based on functional relevance. IEEE Trans. Softw. Eng. 43 (July), 1–22. doi:10.1109/TSE.2016.2645572.

CppRefactory, 2001. Cpprefactory. http://cpptool.sourceforge.net/.

Dig, D., 2017. The landscape of refactoring research in the last decade (keynote). SIGPLAN Not. 52 (12). doi:10.1145/3170492.3148040. 1–1

Easterbrook, S., Singer, J., Storey, M.-A., Damian, D., 2008. Selecting Empirical Methods for Software Engineering Research. Springer London, London, pp. 285–311.

Elssamadisy, A., 2007. Patterns of Agile Practice Adoption: The Technical Cluster. C4Media.

Express, D., 2018. Coderush: Ide Productivity Tools for Visualstudio. http://www.devexpress.com/Products/CodeRush/

Fanta, R., Rajlich, V., 1998. Reengineering object-oriented code. In: Proceedings of the 15th IEEE International Conference on Software Maintenance (ICSM'98). IEEE Computer Society, Washington, DC, USA, pp. 238–246.

Feathers, M., 2004. Working Effectively with Legacy Code. Prentice Hall.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., Figueiredo, E., 2016. A review-based comparative study of bad smell detection tools. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering. Association for Computing Machinery, New York, NY, USA, p. 1. doi:10.1145/2915970.2915984.

Fowler, M., Beck, K., 2019. Refactoring: Improving the Design of Existing Code - Second Edition. Pearson.

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Garousi, V., Fernandes, J.M., 2016. Highly-cited papers in software engineering: the top-100. Inf. Softw. Technol. 71, 108–128.

Garousi, V., Fernandes, J.M., 2016. Highly-cited papers in software engineering: the top-100. Inf. Softw. Technol. 71, 108–128. doi:10.1016/j.infsof.2015.11.003.

Garousi, V., Giray, G., Tuzun, E., Catal, C., Felderer, M., 2019. Closing the gap between software engineering education and industrial needs. IEEE Software doi:10.1109/MS.2018.2880823. 1–1

Garousi, V., Küçük, B., 2018. Smells in software test code: a survey of knowledge in industry and academia. J. Syst. Softw. 138, 52–81. doi:10.1016/j.jss.2017.12.013.

Garousi, V., Mäntylä, M.V., 2016. A systematic literature review of literature reviews in software testing. Inf. Softw. Technol. 80, 195–216.

Garousi, V., Pfahl, D., Fernandes, J.M., Felderer, M., Mäntylä, M.V., Shepherd, D., Arcuri, A., Coşkunçay, A., Tekinerdogan, B., 2019. Characterizing industry-academia collaborations in software engineering: evidence from 101 projects. Empir. Softw. Eng. doi:10.1007/s10664-019-09711-y.

Garousi, V., Shepherd, D.C., Herkiloglu, K., 2019. Successful engagement of practitioners and software engineering researchers: Evidence from 26 international industry-academia collaborative projects. IEEE Software doi:10.1109/MS.2019.2914663. 1–1

Goldman, A., Kon, F., Silva, P.J.S., Yoder, J.W., 2004. Being extreme in the classroom: experiences teaching xp. J. Braz. Comput. Soc. 10 (2), 4–20. doi:10.1007/BF03192356.

Goues, C.L., Jaspan, C., Ozkaya, I., Shaw, M., Stolee, K.T., 2018. Bridging the gap: from research to practical advice. IEEE Softw. 35 (5), 50–57. doi:10.1109/MS.2018.3571235.

GRADIŠNIKMITJA and HERIČKO, M., 2018. Impact of code smells on the rate of defects in software: a literature review. In: SQAMIA 2018: 7th Workshop of Software Quality, pp. 1–21. doi:10.20944/preprints201810.0059.v1.

Griffith, I., Wahl, S., Izurieta, C., 2011. Truerefactor : an automated refactoring tool to improve legacy system and application comprehensibility. In: Proceedings of the 24th International Conference on Computer Applications in Industry and Engineering (CAINE), p. 1.

Griswold, W.G., Opdyke, W.F., 2015. The birth of refactoring: a retrospective on the nature of high-impact software engineering research. IEEE Softw. 32 (6), 30–38. doi:10.1109/MS.2015.107.

Gupta, A., Suri, B., Misra, S., 2017. A systematic literature review: code bad smells in java source code. In: ICCSA 2017, pp. 665–682. doi:10.1007/978-3-319-62404-4_49.

Haendler, T., Frysak, J., 2018. Deconstructing the refactoring process from a problem-solving and decision-making perspective. In: Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018), pp. 363–372. doi:10.5220/0006915903630372.

Haendler, T., Neumann, G., 2019. Serious refactoring games. In: 52nd Hawaii International Conference on System Sciences (HICSS-52), p. 1.

Haendler, T., Neumann, G., Smirnov, F., 2019. An interactive tutoring system for training software refactoring. In: 11th International Conference on Computer Supported Education (CSEDU), 1, p. 1.

Hall, T., Zhang, M., Bowes, D., Sun, Y., 2014. Some code smells have a significant but small effect on faults. ACM Trans. Softw. Eng. Methodol. 23 (4), 33:1–33:39. doi:10.1145/2629648.

Highsmith, J., Fowler, M., 2001. The agile manifesto. Softw. Dev. Mag. 9 (8), 29–30.

Hoda, R., Salleh, N., Grundy, J., Tee, H.M., 2017. Systematic literature reviews in agile software development: a tertiary study. Inf. Softw. Technol. 85, 60–70.

Hozano, M., Garcia, A., Fonseca, B., Costa, E., 2018. Are you smelling it? Investigating how similar developers detect code smells. Inf. Softw. Technol. 93, 130–146.
Imtiaz, S., Bano, M., Ikram, N., Niazi, M., 2013. A tertiary study: experiences of conducting systematic literature reviews in software engineering. In: Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering. Association for Computing Machinery, New York, NY, USA, pp. 177–182. doi:10.1145/2460999.2461025.
International Standards Organisation (ISO), 1991. International Standard ISO/IEC 9126. Information Technology: Software Product Evaluation: Quality Characteristics and Guidelines for their Use.
Jemerov, D., 2008. Implementing refactorings in intellij idea. In: Proceedings of the 2nd Workshop on Refactoring Tools. Association for Computing Machinery, New York, NY, USA, p. 1. doi:10.1145/1636642.1636655.
JetBrains, 2018. The Most Intelligent Extension for Visual Studio :: Resharper - c, vb.net, linq, asp.net, asp.net mvc, xaml, xml, javascript, html, build scripts. best-of-breed tools for code refactoring, code quality analysis, code cleanup, navigation, code generation, unit testing, and code templates. http://www.jetbrains.com/resharper/index.html
JRefactory, 2018. Jrefactory. http://jrefactory.sourceforge.net/
Kerievsky, J., 2004. Refactoring to Patterns. Addison-Wesley.
Khan, M.U., Sherin, S., Iqbal, M.Z., Zahid, R., 2019. Landscaping systematic mapping studies in software engineering: a tertiary study. J. Syst. Softw. 149, 396–436. doi:10.1016/j.jss.2018.12.018.
Khomh, F., Di Penta, M., Gueheneuc, Y., 2009. An exploratory study of the impact of code smells on software change-proneness. In: 2009 16th Working Conference on Reverse Engineering, pp. 75–84. doi:10.1109/WCRE.2009.28.
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. Empir. Softw. Eng. 17 (3), 243–275. doi:10.1007/s10664-011-9171-y.
Kitchenham, B., Brereton, O.P., Budgen, D., Turner, M., Bailey, J., Linkman, S., 2009. Systematic literature reviews in software engineering – a systematic literature review. Inf. Softw. Technol. 51 (51), 7–15.
Kitchenham, B., Brereton, P., Budgen, D., 2010. The educational value of mapping studies of software engineering literature. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1. ACM, New York, NY, USA, pp. 589–598. doi:10.1145/1806799.1806887.
Kitchenham, B., Charters, S., Budgen, D., Brereton, P., Turner, M., Linkman, S., 2007. Guidelines for Performing Systematic Literature Reviews in Software Engineering.
Kitchenham, B., Pretorius, R., Budgen, D., Brereton, O.P., Turner, M., Niazi, M., Linkman, S., 2010. Systematic literature reviews in software engineering – a tertiary study. Inf. Softw. Technol. 52 (8), 792–805. doi:10.1016/j.infsof.2010.03.006.
Koschke, R., 2007. Survey of research on software clones. In: Koschke, R., Merlo, E., Walenstein, A. (Eds.), Duplication, Redundancy, and Similarity in Software. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany, p. 1.
Kruchten, P., Nord, R.L., Ozkaya, I., 2012. Technical debt: from metaphor to theory and practice. IEEE Softw. 29 (6), 18–21. doi:10.1109/MS.2012.167.
Lacerda, G., 2019. Online Repository for Tertiary Systematic Review about Code Smells and Refactoring (Package Replication). https://bit.ly/2WRD1N2
Laguna, M.A., Crespo, Y., 2013. A systematic mapping study on software product line evolution: from legacy system reengineering to product line refactoring. Sci. Comput. Program. 78 (8), 1010–1034. doi:10.1016/j.scico.2012.05.003.
Lahtinen, S., Leppänen, M., 2016. Refactoring patterns, practices for daily work. In: Proceedings of the 10th Travelling Conference on Pattern Languages of Programs. ACM, New York, NY, USA, pp. 6:1–6:8. doi:10.1145/3022636.3022642.
Lee, Y.Y., Harwell, S., Khurshid, S., Marinov, D., 2013. Temporal code completion and navigation. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 1181–1184. doi:10.1109/ICSE.2013.6606673.
Leppänen, M., Lahtinen, S., Kuusinen, K., Mäkinen, S., Männistö, T., Itkonen, J., Yli-Huumo, J., Lehtonen, T., 2015. Decision-making framework for refactoring. In: 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), pp. 61–68. doi:10.1109/MTD.2015.7332627.
Leppänen, M., Mäkinen, S., Lahtinen, S., Sievi-Korte, O., Tuovinen, A., Männistö, T., 2015. Refactoring - a shot in the dark? IEEE Softw. 32 (6), 62–70. doi:10.1109/MS.2015.132.
Li, H., Thompson, S., 2012. A domain-specific language for scripting refactorings in erlang. In: de Lara, J., Zisman, A. (Eds.), Fundamental Approaches to Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 501–515.
Li, H., Thompson, S., Lövei, L., Horváth, Z., Kozsik, T., Víg, A., Nagy, T., 2006. Refactoring erlang programs. In: The Proceedings of 12th International Erlang/OTP User Conference, p. 1. Stockholm, Sweden
Li, H., Thompson, S., Orosz, G., Töth, M., 2008. Refactoring with wrangler, updated: Data and process refactorings, and integration with eclipse. In: Horváth, Z., Teoh, T. (Eds.), Proceedings of the Seventh ACM SIGPLAN Erlang Workshop. ACM Press, NY, USA, p. 12.
Li, Z., Avgeriou, P., Liang, P., 2015. A systematic mapping study on technical debt and its management. J. Syst. Softw. 101, 193–220. doi:10.1016/j.jss.2014.12.027.
López, C., Alonso, J.M., Marticorena, R., Maudes, J.M., 2014. Design of e-activities for the learning of code refactoring tasks. In: 2014 International Symposium on Computers in Education (SIIE), pp. 35–40. doi:10.1109/SIIE.2014.7017701.
Man, B.R., 2018. Bicycle Repair Man. http://bicyclerepair.sourceforge.net/
Mantyla, M., 2003. Bad Smells in Software - A Taxonomy and an Empirical Study. Helsinki University of Technology.
Mantyla, M., Vanhanen, J., Lassenius, C., 2003. A taxonomy and an initial empirical study of bad smells in code. In: Proceedings of the 19th IEEE International Conference on Software Maintenance (ICSM'03). IEEE Computer Society, Washington, DC, USA. 381–
Mariani, T., Vergilio, S.R., 2017. A systematic review on search-based refactoring. Inf. Softw. Technol. 83, 14–34. doi:10.1016/j.infsof.2016.11.009.
Martin, R.C., 2008. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall.
Martin, R.C., Martin, M., 2007. Agile Principles, Patterns, and Practices in C. Prentice Hall.
McCall, J., 1977. Factors in Software Quality: Preliminary Handbook on Software Quality for an Acquisiton Manager, 1–3. General Electric.
Mealy, E., Carrington, D., Strooper, P., Wyeth, P., 2007. Improving usability of software refactoring tools. In: 2007 Australian Software Engineering Conference (ASWEC'07), pp. 307–318. doi:10.1109/ASWEC.2007.24.
Mealy, E., Strooper, P., 2006. Evaluating software refactoring tool support. In: Australian Software Engineering Conference (ASWEC'06), pp. 10–340. doi:10.1109/ASWEC.2006.26.
Mens, T., Demeyer, S., Bois, B.D., Stenten, H., Gorp, P.V., 2003. Refactoring: Current research and future trends. Electronic Notes in Theoretical Computer Science 82 (3), 483–499. doi:10.1016/S1571-0661(05)82624-6. LDTA'2003 - Language descriptions, Tools and Applications
Mens, T., Tourwe, T., 2004. A survey of software refactoring. IEEE Trans. Softw. Eng. 30 (2), 126–139. doi:10.1109/TSE.2004.1265817.
Mihancea, P.F., 2006. Towards a client driven characterization of class hierarchies. In: 14th IEEE International Conference on Program Comprehension (ICPC'06). IEEE, pp. 285–294. doi:10.1109/ICPC.2006.48.
Mihancea, P.F., Marinescu, R., 2009. Discovering comprehension pitfalls in class hierarchies. In: 2009 13th European Conference on Software Maintenance and Reengineering. IEEE, pp. 7–16. doi:10.1109/CSMR.2009.31.
Misbhauddin, M., Alshayeb, M., 2015. UML model refactoring: a systematic literature review. Empir. Softw. Eng. 20 (1), 206–251. doi:10.1007/s10664-013-9283-7.
Moha, N., Guéhéneuc, Y.-G., Duchien, L., Meur, A.-F.L., 2010. Decor: a method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng..
Mohan, M., Greer, D., 2018. A survey of search-based refactoring for software maintenance. J. Softw. Eng. Res.Dev. 6 (1), 3. doi:10.1186/s40411-018-0046-4.
Murphy-Hill, E., 2006. Improving usability of refactoring tools. In: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications. ACM, New York, NY, USA, pp. 746–747. doi:10.1145/1176617.1176705.
Murphy-Hill, E., Black, A., 2007. Why don't people use refactoring tools? In: Proceedings of the 1st Workshop on Refactoring Tools (WRT'07), p. 1. Berlin, Germany
Murphy-Hill, E., Black, A., 2008. Seven habits of highly effective smell detector. In: Proceedings of the International Workshop on Recommendation Systems for Software Engineering (RSSE'08). ACM, New York, NY, USA, pp. 36–40.
Murphy-Hill, E., Black, A.P., 2007. High velocity refactorings in eclipse. In: Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology EXchange. Association for Computing Machinery, New York, NY, USA, pp. 1–5. doi:10.1145/1328279.1328280.
Murphy-Hill, E., Black, A.P., 2010. An interactive ambient visualization for code smells. In: Proceedings of the 5th International Symposium on Software Visualization. Association for Computing Machinery, New York, NY, USA, pp. 5–14. doi:10.1145/1879211.1879216.
Murphy-Hill, E., Parnin, C., Black, A.P., 2012. How we refactor, and how we know it. IEEE Trans. Softw. Eng. 38 (1), 5–18. doi:10.1109/TSE.2011.41.
Nurdiani, I., Börstler, J., Fricker, S.A., 2016. The impacts of agile and lean practices on project constraints: a tertiary study. J. Syst. Softw. 119, 162–183.
Offutt, J., Alexander, R., Wu, Y., Xiao, Q., Hutchinson, C., 2001. A fault model for subtype inheritance and polymorphism. In: Proceedings 12th International Symposium on Software Reliability Engineering. IEEE Comput. Soc, pp. 84–93. doi:10.1109/ISSRE.2001.989461.
Opdyke, W., 1992. Refactoring Object-Oriented Frameworks. University of Illinois at Urbana-Champaign.
d. P. Sobrinho, E.V., De Lucia, A., d. A. Maia, M., 2018. A systematic literature review on bad smells – 5 w's: which, when, what, who, where. IEEE Transactions on Software Engineering doi:10.1109/TSE.2018.2880977. 1–1
Parnin, C., Görg, C., Nnadi, O., 2008. A catalogue of lightweight visualizations to support code smell inspection. In: Proceedings of the 4th ACM Symposium on Software Visualization. Association for Computing Machinery, New York, NY, USA, pp. 77–86. doi:10.1145/1409720.1409733.
Pate, J.R., Tairas, R., Kraft, N.A., 2013. Clone evolution: a systematic review. J. Softw. 25 (3), 261–283. doi:10.1002/smr.579.
Perez, J., 2011. Refactoring Planning for Design Smell Correction in Object-Oriented Software. Escuela Técnica Superior e Ingeniería Informática - Universidad de Valladolid.
Petersen, K., Feldt, R., Mujtaba, S., Mattsson, M., 2008. Systematic mapping studies in software engineering. In: Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering. BCS Learning & Development Ltd., Swindon, UK, pp. 68–77.
Petersen, K., Vakkalanka, S., Kuzniarz, L., 2015. Guidelines for conducting systematic mapping studies in software engineering: an update. Inf. Softw. Technol. 64, 1–18. doi:10.1016/j.infsof.2015.03.007.
Piveta, E.K., 2009. Improving the Search for Refactoring Opportunities on Object-Oriented and Aspect-Oriented Software. PPGC - Universidade Federal do Rio Grande do Sul (UFRGS).
Piveta, E.K., Hecht, M., Pimenta, M.S., Price, R., 2006. Detecting bad smells in aspectj. JUCS 12 (7), 811–827.

Rasool, G., Arshad, Z., 2015. A review of code smell mining techniques. J. Softw. 27 (11), 867–895. doi:10.1002/smr.1737.
Rattan, D., Bhatia, R., Singh, M., 2013. Software clone detection: a systematic review. Inf. Softw. Technol. 55 (7), 1165–1199. doi:10.1016/j.infsof.2013.01.008.
Rattan, D., Kaur, J., 2016. Systematic mapping study of metrics based clone detection techniques. In: Proceedings of the International Conference on Advances in Information Communication Technology & Computing - AICTC '16, pp. 1–7. doi:10.1145/2979779.2979855.
Refactory, 2013. Refactory. http://www.modelrefactoring.org/index.php/Refactoring
Riel, A.J., 1996. Object-Oriented Design Heuristics, 1st Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
Rios, N., de MendonÃ§a Neto, M.G., SpÃnola, R.O., 2018. A tertiary study on technical debt: types, management strategies, research trends, and base information for practitioners. Inf. Softw. Technol. 102, 117–145.
Roberts, D.B., 1999. Practical Analysis for Refactoring. University of Illinois at Urbana-Champaign.
Rochimah, S., Arifiani, S., Insanittaqwa, V.F., 2015. Non-Source code refactoring: A Systematic literature review. Int. J. Softw. Eng. and Its Applications 9 (6), 197–214. doi:10.14257/ijseia.2015.9.6.19.
Rope, 2018. Rope Python Refactoring Library.... https://github.com/python-rope/rope
Roy, C.K., Zibran, M.F., Koschke, R., 2014. The vision of software clone management: Past, present, and future (Keynote paper). In: 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE). IEEE, pp. 18–33. doi:10.1109/CSMR-WCRE.2014.6747168.
Saadeh, E., Kourie, D.G., 2009. Composite refactoring using fine-grained transformations. In: Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists. Association for Computing Machinery, New York, NY, USA, pp. 22–29. doi:10.1145/1632149.1632154.
Sabir, F., Palma, F., Rasool, G., Guéhéneuc, Y.-G., Moha, N., 2018. A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems. Software 49 (August), 1–37. doi:10.1002/spe.2639.
Sandalski, M., Stoyanova-Doycheva, A., Popchev, I., Stoyanov, S., 2011. Development of a refactoring learning environment. Cybern. Inf. Technol. 11 (2).
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. J. Syst. Softw. 144 (July), 450–477. doi:10.1016/j.jss.2018.07.035.
dos Santos Neto, B.F., Ribeiro, M., da Silva, V.T., Braga, C., de Lucena, C.J.P., de Barros Costa, E., 2015. Autorefactoring: a platform to build refactoring agents. Expert Syst. Appl. 42 (3), 1652–1664.
Sharma, T., Spinellis, D., 2018. A survey on software smells. J. Syst. Softw. 138, 158–173. doi:10.1016/j.jss.2017.12.034.
Sheikh, J.A., Fields, B., Duncker, E., 2011. The cultural integration of knowledge management into interactive design. In: Smith, M.J., Salvendy, G. (Eds.), Human Interface and the Management of Information. Interacting with Information. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 48–57.
Sheneamer, A., Kalita, J., 2016. A survey of software clone detection techniques. Int. J. Comput. Appl. 137 (10), 975–8887. doi:10.1109/MITICON.2016.8025227.
Sidhu, B.K., Singh, K., Sharma, N., 2018. Refactoring UML models of object-oriented software: a systematic review. Int. J. Softw. Eng. Knowl. Eng. 28 (9), 1287–1319. doi:10.1145/1878431.1878433.
da Silva, F.Q., Santos, A.L., Soares, S., França, A.C.C., Monteiro, C.V., Maciel, F.F., 2011. Six years of systematic literature reviews in software engineering: An updated tertiary study. Information and Software Technology 53 (9), 899–913. doi:10.1016/j.infsof.2011.04.004. Studying work practices in Global Software Engineering
da Silva, F.Q.B., Santos, A.L.M., Soares, S.C.B., França, A.C.C., Monteiro, C.V.F., 2010. A critical appraisal of systematic reviews in software engineering from the perspective of the research questions asked in the reviews. In: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. ACM, New York, NY, USA, pp. 33:1–33:4. doi:10.1145/1852786.1852830.
Singh, R., Kumar, A., 2018. Identifying various code-smells and refactoring opportunities in object-oriented software system : a systematic literature review. Int. J. Fut. Revol.Comput. Sci. Commun. Eng. 8 (March), 62–74.
Singh, S., Kaur, S., 2017. A systematic literature review: refactoring for disclosing code smells in object oriented software. Ain Shams Eng. J. doi:10.1016/j.asej.2017.03.002.
Sjøberg, D.I.K., Yamashita, A., Anda, B.C.D., Mockus, A., Dybå, T., 2013. Quantifying the effect of code smells on maintenance effort. IEEE Trans. Softw. Eng. 39 (8), 1144–1156. doi:10.1109/TSE.2012.89.
Smith, S., Stoecklin, S., Serino, C., 2006. An innovative approach to teaching refactoring. In: Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education. ACM, New York, NY, USA, pp. 349–353. doi:10.1145/1121341.1121451.
Society, I.C., Bourque, P., Fairley, R.E., 2014. Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0, 3rd IEEE Computer Society Press, Los Alamitos, CA, USA.
Software, W.T., 2018. Visual Assist - A Visualstudio Extension by Whole Tomato Software. http://www.wholetomato.com/
Sousa, B.L., Bigonha, M.A.S., Ferreira, K.A.M., 2018. A systematic literature mapping on the relationship between design patterns and bad smells. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing - SAC '18. ACM Press, pp. 1528–1535. doi:10.1145/3167132.3167295.
Stoecklin, S., Smith, S., Serino, C., 2007. Teaching students to build well formed object-oriented methods through refactoring. In: Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education. ACM, New York, NY, USA, pp. 145–149. doi:10.1145/1227310.1227364.
Taibi, D., Janes, A., Lenarduzzi, V., 2017. How developers perceive smells in source code: areplicated study. Inf. Softw. Technol. 92, 223–235.
Telea, A., Voinea, L., 2011. Visual software analytics for the build optimization of large-scale software systems. Comput. Stat. 26 (4), 635–654.
Tempero, E., Gorschek, T., Angelis, L., 2017. Barriers to refactoring. Commun. ACM 60 (10), 54–61. doi:10.1145/3131873.
Terra, R., Valente, M.T., Miranda, S., Sales, V., 2018. JMove: a novel heuristic and tool to detect move method refactoring opportunities. J. Syst. Softw. 138, 19–36. doi:10.1016/j.jss.2017.11.073.
Tsantalis, N., 2010. Evaluation and Improvement of Software Architecture: Identification of Design Problems in Object-Oriented Systems and Resolution through Refactorings. Macedonia University.
Tsantalis, N., 2018. Jdeodorant. https://github.com/tsantalis/JDeodorant
Tsantalis, N., Chatzigeorgiou, A., 2011. Ranking refactoring suggestions based on historical volatility. In: 2011 15th European Conference on Software Maintenance and Reengineering, pp. 25–34. doi:10.1109/CSMR.2011.7.
University of York, 2018. Centre for Reviews and Dissemination. https://www.york.ac.uk/crd/
Vakilian, M., Chen, N., Negara, S., Rajkumar, B.A., Bailey, B.P., Johnson, R.E., 2012. Use, disuse, and misuse of automated refactorings. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 233–243. doi:10.1109/ICSE.2012.6227190.
Vale, G., Figueiredo, E., Abilio, R., Costa, H., 2014. Bad smells in software product lines: a systematic review. In: 2014 Eighth Brazilian Symposium on Software Components, Architectures and Reuse. IEEE, pp. 84–94. doi:10.1109/SBCARS.2014.21.
Vedurada, J., Nandivada, V.K., 2017. Refactoring opportunities for replacing type code with state and subclass. In: Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017, pp. 305–307. doi:10.1109/ICSE-C.2017.97.
Wake, W.C., 2003. Refactoring Workbook. Addison-Wesley.
Webster, B.F., 1995. Pitfalls of Object Oriented Development. M and T Books, New York, NY, USA.
Welf Löwe, W., Panas, T., 2005. Rapid construction of software comprehension tools. Int. J. Software Eng. Knowl. Eng. 15 (6), 995–1025.
Wohlin, C., 2007. An analysis of the most cited articles in software engineering journals - 2000. Information and Software Technology 49 (1), 2–11. doi:10.1016/j.infsof.2006.08.004. Most Cited Journal Articles in Software Engineering - 2000
Wohlin, C., 2008. An analysis of the most cited articles in software engineering journals – 2001. Information and Software Technology 50 (1), 3–9. doi:10.1016/j.infsof.2007.10.002. Special issue with two special sections. Section 1: Most-cited software engineering articles in 2001. Section 2: Requirement engineering: Foundation for software quality
Wohlin, C., 2014. 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. ACM, New York, NY, USA, pp. 38:1–38:10.
XRefactory, 1998. Xrefactory. http://www.xref.sk/about.html.
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. IEEE Press, Piscataway, NJ, USA, pp. 682–691.
Zazworka, N., Seaman, C., Shull, F., 2011. Prioritizing design debt investment opportunities. In: Proceedings of the 2nd Workshop on Managing Technical Debt. Association for Computing Machinery, New York, NY, USA, pp. 39–42. doi:10.1145/1985362.1985372.
Zhang, M., Baddoo, N., Wernick, P., Hall, T., 2008. Improving the precision of fowler's definitions of bad smells. In: 2008 32nd Annual IEEE Software Engineering Workshop, pp. 161–166. doi:10.1109/SEW.2008.26.
Zhang, M., Hall, T., Baddoo, N., 2011. Code bad smells: a review of current knowledge. Journal of Software Maintenance and Evolution: Research and Practice 23 (3), 179–202. doi:10.1002/smr.521.

Guilherme Lacerda is Ph.D. student in computer science at UFRGS (Brazil), studying about software quality, software maintenance and evolution, smells, and refactoring. Book author, speaker, university lecturer (Unisinos), and associate consultant (Wildtech). Free/Libre/Open Source Software (FLOSS) movement enthusiast. Creator of DR-Tools Suite (http://drtools.site)

Fabio Petrillo is associate professor in the Department of Computer Sciences and Mathematics (DIM) at Université du Québec à Chicoutimi (Canada) since 2018. His main goal is to create theories and techniques to improve software engineers' life. In his research career, he has worked on Empirical Software Engineering, Software Quality, Debugging, Service-Oriented Architecture, Cloud Computing, and Agile Methods. He has been recognized as a pioneer and an international reference on Software Engineering for Computer Games. He is the creator of Swarm Debugging, a new collaborative approach to support debugging activities.

Marcelo S. Pimenta is full professor at Institute of Informatics (INF), Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, south of Brazil. He received his PhD in Computer Science from Université Toulouse 1 (France) in 1997 - at Laboratoire LIIHS "Logiciel Interactif et Interaction Homme-Système" (Interactive Software and Human Computer Interaction team) of University of Toulouse 1 Capitole,

Toulouse, France - and the bachelor and master's degree in Computer Science from UFRGS in 1988 and 1991, respectively. He is head of Laboratório de Computação Musical (LCM), the UFRGS Computer Music Laboratory. Since 1998, he is member of a multidisciplinary research group at INF/UFRGS working with topics in Human-Computer Interaction, Software Engineering, and Computer Music with emphasis in the integration of these areas. Currently his research is focused on adaptive interfaces, networked music, ubiquitous music, synergistic modeling, human aspects of software development, usercentered software engineering and, more recently, new forms of governance and the delivery of public services with the support of Information Technologies. Before (from 1990 until 1998), he was lecturer in the Departamento de Informática and Estatística (INE) of the Federal University of Santa Catarina (UFSC), Florianópolis, Brazil and vice-coordinator of LabiUtil - a Brazilian pioneer Usability Lab.

Yann-Gaël Guéhéneuc is full professor at the Department of Computer Science and Software Engineering of Concordia University since 2017, where he leads the Ptidej team on evaluating and enhancing the quality of the software systems, focusing on the Internet of Things and researching new theories, methods, and tools to understand, evaluate, and improve the development, release, testing, and security of such systems. Prior, he was faculty member at Polytechnique Montréal and Université de Montréal, where he started as assistant professor in 2003. In 2014, he was awarded the NSERC Research Chair Tier II on Patterns in Mixed-language Systems. In 2013–2014, he visited KAIST, Yonsei U., and Seoul National University, in Korea, as well as the National Institute of Informatics, in Japan, during his sabbatical year. In 2010, he became IEEE Senior Member. In 2009, he obtained the NSERC Research Chair Tier II on Software Patterns and Patterns of Software. In 2003, he received a Ph.D. in Software Engineering from University of Nantes, France, under Professor Pierre Cointe's supervision. His Ph.D. thesis was funded by Object Technology International, Inc. (now IBM Ottawa Labs.), where he worked in 1999 and 2000. In 1998, he graduated as engineer from École des Mines of Nantes. His research interests are program understanding and program quality, in particular through the use and the identification of recurring patterns. He was the first to use explanation-based constraint programming in the context of software engineering to identify occurrences of patterns. He is interested also in empirical software engineering; he uses eye-trackers to understand and to develop theories about program comprehension. He has published papers in international conferences and journals, including IEEE TSE, Springer EMSE, ACM/IEEE ICSE, IEEE ICSME, and IEEE SANER. He was the program cochair and general chair of several events, including IEEE SANER'15, APSEC'14, and IEEE ICSM'13.